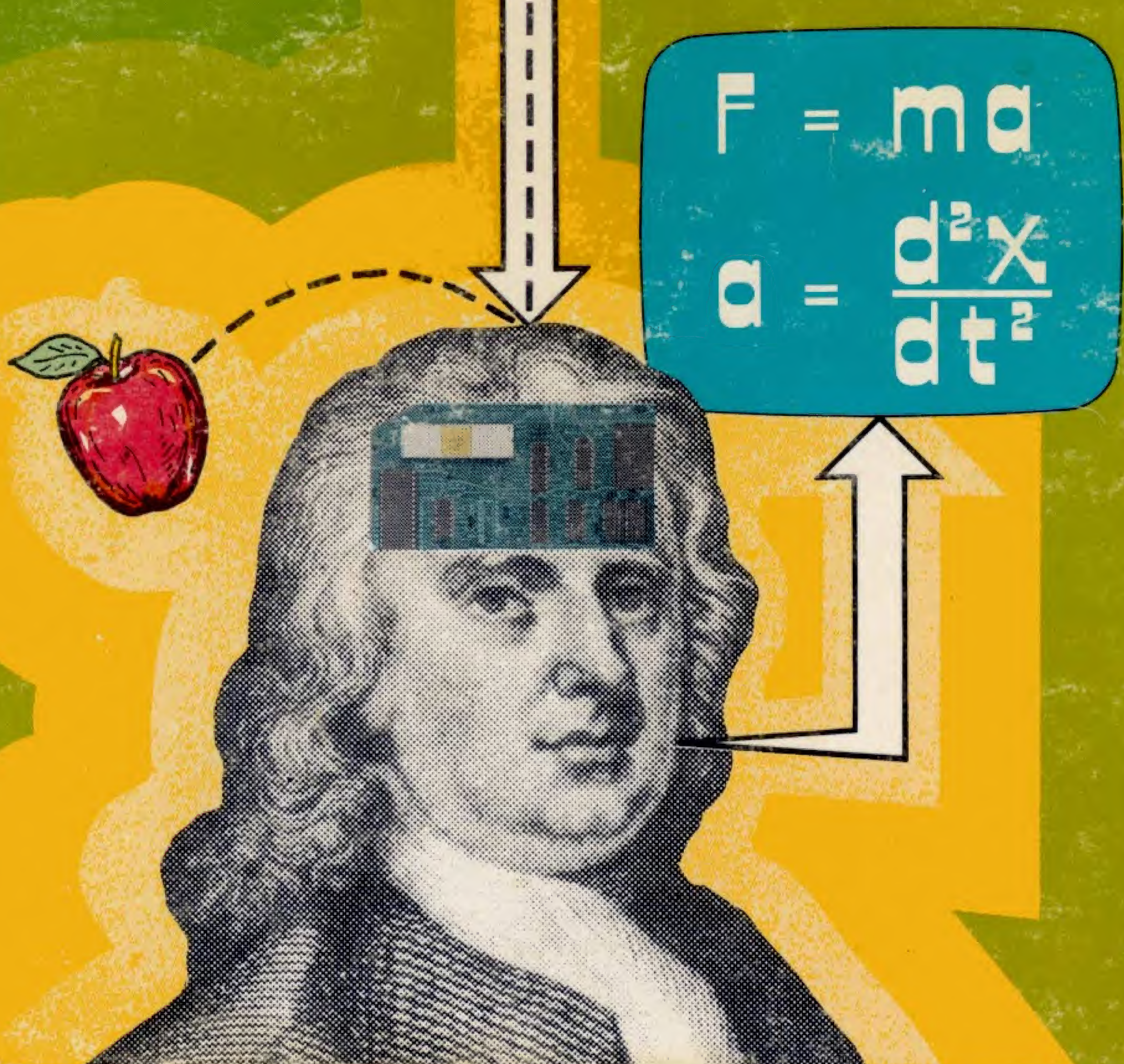
21500

the Z-80 microcomputer handbook

William Barden, Jr.



The Z-80 Microcomputer Handbook

by

William Barden, Jr.

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

Copyright © 1978 by Howard W. Sams & Co., Inc.,
Indianapolis, Indiana 46268

FIRST EDITION
THIRD PRINTING—1979

All rights reserved. Reproduction or use, without express permission, of editorial or pictorial content, in any manner, is prohibited. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-21500-4
Library of Congress Catalog Card Number: 77-93166

Printed in the United States of America.

Preface

Microprocessors have evolved from units that handled data in 4-bit slices with rudimentary instruction sets into devices that rival, or surpass, minicomputers in architecture and software instruction repertoire. The Zilog Model Z-80 represents a microprocessor that is extremely sophisticated from both a hardware implementation and software implementation viewpoint. The Z-80 microprocessor is truly a computer on a chip that requires only a few external components—a 5-volt power supply, a simple oscillator, and read-only memory—to construct a complete computer system. The instruction set of the Z-80 includes that of the Intel 8080A as a subset, making the Z-80 an ideal software replacement for the 8080A; the Z-80 has many new instructions and addressing modes to supplement the 8080A instructions. A search of a string of characters, for example, can be implemented with one instruction after initialization, the one search instruction replacing four equivalent instructions in other microprocessors.

In addition to the Z-80 microprocessor itself, Zilog has implemented other devices to supplement the power of the Z-80. A PIO provides parallel I/O with two 8-bit ports, software configured I/O, vectored-interrupt capability, and automatic priority interrupt encoding. A CTC, or Counter-Timer-Circuit, provides programmable counting and timing functions for real-time events. Other major devices are also available. Zilog and other manufacturers have developed microcomputer systems based on this family of Z-80 devices, and the systems have played their role in narrowing the gap between "minicomputer systems" and "microcomputer systems," a division that becomes less and less distinct from month to month.

The purpose of this book is threefold, to acquaint the reader with the hardware of the Z-80, to discuss the almost overwhelming (in number of instructions) software aspects of the Z-80, and to describe microcomputer systems built around the Z-80.

Section I discusses Z-80 hardware. The architecture, interface signals, and timing are discussed in the first two chapters. Addressing modes and instructions are covered in the next two chapters; both addressing and instruction repertoire are fairly easily grouped and explained, although they may appear confusing at first glance. The effect of arithmetic operations and other operations on CPU flags is presented in Chapter 6. The powerful interrupt sequences of the Z-80 are discussed in the next chapter. Chapter 8 describes interfacing examples of I/O and memory devices.

Section II describes Z-80 software. A representative Z-80 assembler program is introduced in the first chapter of the section. An assembler is almost a necessity with a microprocessor having such a large instruction set, but machine language aspects are also covered. Chapters 10 through 15 present the common programming operations of moving data, arithmetic operations, shifting and bit operations, list and table procedures, subroutine use, and I/O functions in relation to instruction set groups. Many examples of each kind of operation are provided. The last chapter of the section details some commonly used subroutines written in Z-80 assembly language.

The third section discusses microcomputers built around the Z-80. Chapter 17 covers Zilog products including the microcomputer board products in the Z-80 family and development systems. Four other Z-80 microcomputer manufacturers are described in the last chapter. Technical Design Labs, Inc., Cromemco, Inc., The Digital Group, Inc., and Radio Shack. The hardware and software aspects of all five manufacturers are presented.

The Z-80 will prove attractive to many users, not only as a successor to the 8080A, but as a powerful computer in its own right.

The Z-80 will soon have a successor, in this dynamic microcomputer development environment, but for the time being it represents microcomputer "state-of-the-art." The author hopes that the reader will derive a great deal of benefit from the book and that the Z-80 will solve a few hardware and software implementation problems.

Much credit for this book goes to my wife, Janet, who has solved my major software implementation problems—manuscript preparation.

WILLIAM BARDEN, JR.

To Bill and Norma and
the Little Green Onions.

Contents

SECTION I–Z-80 Hardware

CHAPTER 1

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

CHAPTER 6

CHAPTER 7

CHAPTER 8

SECTION II–Z-80 Software

CHAPTER 9

CHAPTER 10

CHAPTER 11

CHAPTER 12

CHAPTER 13

CHAPTER 14

CHAPTER 15

CHAPTER 16

SECTION III—Z-80 Microcomputers

CHAPTER 17

CHAPTER 18

APPENDIX A

APPENDIX B

APPENDIX C

APPENDIX D

APPENDIX E

APPENDIX F

SECTION I

Z-80 Hardware

CHAPTER 1

Introduction

In 1971, Intel Corporation introduced the first microcomputer on a chip, the Intel 4004. Although the 4004 was truly not a self-contained computer on a single Large-Scale–Integration (LSI) chip, it contained a great deal of logic associated with computer central processing unit implementation. One LSI chip replaced hundreds of circuits that were to be found in conventional minicomputers at the time. Although the 46-instruction repertoire was not large, it was adequate for control applications which required decision making that could not easily be implemented in programmable-logic arrays and in which extensive mathematical processing was not required. The 4004 handled data 4 bits at a time and could perform 100,000 additions of two 4-bit operands per second.

The next generation of microprocessors from Intel retained the PMOS (P-channel metal-oxide semiconductor) fabrication techniques of the 4004, but offered an 8-bit wide *data bus* and a larger instruction repertoire of 48 instructions. Designated the 8008, the microprocessor had a faster instruction cycle time than the 4004 as data for both instruction execution and decoding and for operands could be handled in 8-bit slices. In addition, the 8008 could address 16,384 memory locations of 8 bits each, contained seven 8-bit registers, had memory stack capability, and had a single-level interrupt capability. The 8008 could perform approximately 80,000 additions of two 8-bit operands per second. The instruction set of the 8008 was not compatible with the 4004.

The 8008 and 4004 had achieved widespread usage through the electronics industry in a very short time after their introduction, primarily because there was little else available in the way of microprocessors. To achieve compatibility with the 8008 insofar as instruc-

tion repertoire, the Intel 8080, introduced in late 1973, included the instruction set of the 8008 and supplemented it with 30 more instructions. Users of the 8008 could now change to a faster, more versatile microprocessor while not discarding 8008 software programs, since all 8008 software would presumably execute on the 8080. The 8080 was an NMOS (N-channel metal-oxide semiconductor) microprocessor that allowed faster clock rates. Additions of two 8-bit operands could now be carried out at rates of 500,000 per second. In addition, all other instruction times were much shorter than the 8008 as the 8080 was built around a 40-pin chip, requiring the CPU to do much less time sharing of the data bus between data transfers and instruction implementation.

The 8080 supplemented the hardware features of the 8008. In place of 16,384 (*16K*) memory addresses, the 8080 could address 65,536 (*64K*). Rather than a limited 7-level memory stack, the 8080 offered a memory stack in external memory itself instead of the CPU. A binary-coded decimal or *bcd* capability was built into the arithmetic and logic unit in the CPU; additions of two bcd operands could now be implemented. Expanded addressing modes to permit direct addressing of external memory was offered. Although the 78 instructions of the 8080 still seemed strange to many programmers, the instruction set decidedly had moved away from one for primarily control applications to one that was more general purpose in nature.

In 1976, Intel brought out several variations on the 8080. The Intel 8085 included a serial input/output capability on the microprocessor chip itself. In addition, the 8085 had a requirement of only a single-phase clock (the 8008 and 8080 were two-phase clocks) and a single 5-volt power supply (the 8008 and 8080 required two and three voltages, respectively). As the number of supporting packages had grown impressively (such chips as a programmable peripheral interface, interrupt controller, and crt controller) Intel provided very powerful computing capability at faster and faster speeds (770,000 8-bit adds per second), while still retaining compatability with existing software written for the 8008 and 8080.

Although the 8085 was an improvement over the 8080 in many features, the instruction set remained very similar to the 8080. Only two new instructions were added, one to read serial and interrupt data, and one to write serial and interrupt data. Many of the inherent inadequacies of the 8008 and 8080 remained.

The Zilog, Inc. Z-80 microprocessor chip has provided another level of sophistication for the widely used 8008/8080 base. Bearing in mind that the super computer of today is the surplus bargain of tomorrow, the Z-80 has supplemented the instruction set and capabilities of the 8080 in the same fashion as the 8080 increased the

capabilities of the 8008. In addition, Zilog has produced a family of support chips that supplement the Z-80. The Z-80 is software compatible with the 8080, allowing existing 8008 and 8080 software to be executed on the Z-80. While the limitations of the 8008 and 8080 instructions and architecture must of necessity be retained in the Z-80, the Z-80 offers new instructions, new addressing modes, and new hardware features that provide more capability and versatility than ever before.

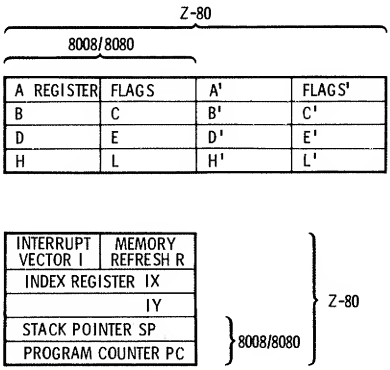


Fig. 1-1. Register comparison 8008, 8080, and Z-80.

In addition to providing the eight 8-bit CPU registers of the 8080, the Z-80 duplicates the eight registers to offer sixteen registers. Two index registers offer indexing capability not provided in the 8080. An interrupt-vector register and memory-refresh register provide special interrupt functions and dynamic memory-refresh capability. Fig. 1-1 shows the basic register arrangement of the 8008, 8080, and Z-80.

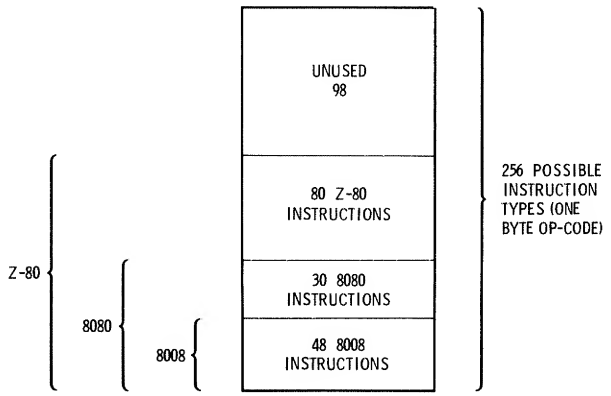


Fig. 1-2. Instruction comparison 8008, 8080, and Z-80.

The 78 instructions of the 8080 are provided in the Z-80, but the total number of instructions comes to 158. Many of these are logical extensions of 8080 instructions, but many are extremely powerful and a complete departure from the 8080. Fig. 1-2 shows the relative differences between the 8008, 8080, and Z-80.

All Input/Output and interrupt capability of the 8080 is retained in the Z-80. I/O is expanded, however, to operate from any CPU register and to operate in "block" fashion, that is, to facilitate transfer of many bytes at a time over a programmed (non-DMA) I/O channel. Interrupts include the standard external interrupt capability of the 8080, but supplement this with a separate "nonmaskable" interrupt similar to the Motorola MC6800 and MOS Technology MCS 6502. Other interrupt capability allows for interrupt vectoring anywhere in memory, rather than just to eight locations in page 0, and for up to 128 levels of interrupts, rather than eight.

The Z-80 Microcomputer Handbook is divided into three sections. Section I covers the hardware aspects of the Z-80. Architecture, interface signals and timing, addressing modes, instruction set, flags, interrupt sequences, interface of memory and I/O devices, and DMA operation are discussed. When applicable, differences between the 8080 and Z-80 are discussed. Section II discusses Z-80 software, grouped in similar manner to Zilog Z-80 documentation. Section II also provides programming examples of Z-80 code. Many times, a short section of a program will greatly clarify the somewhat pedantic descriptions of individual instructions. Section III discusses five microcomputer manufacturers that have built microcomputers around the Z-80 microprocessor chip. Appendix A provides complete electrical specifications for the Z-80. Appendix B cross-references 8080 instructions to the Z-80 instruction set and Appendix C provides a short description of each Z-80 instruction. Appendix D reviews binary and hexadecimal representation while Appendix E lists ASCII character codes. The last appendix, Appendix F, lists Z-80 Microcomputer manufacturers.

CHAPTER 2

Z-80 Architecture

The architecture of the Z-80 is shown in Fig. 2-1. Thirteen CPU and system control signals are sent to or generated in the instruction decode and CPU control portion of the microprocessor. The data bus is eight bits wide and is the path for all data transferred between external memory and input/output devices and CPU registers. The address bus is sixteen bits wide. Normally the address bus would specify an external memory address of 0 to 65535 (0 to 64K − 1) since the Z-80 has a full complement of input/output instructions and no "memory-mapped" input/output would be required. (In memory-mapped input/output, a portion of the memory addresses must be dedicated to addresses of input/output devices).

The main path for data within the CPU is an internal data bus which connects the CPU registers, arithmetic and logical unit, data bus control, and instruction register. The arithmetic and logical unit performs addition, subtraction, logical functions of ANDing, ORing, and exclusive ORing, and shifting operations between two 8-bit operands. In addition, binary-coded decimal (bcd) operations may be performed under control of a Decimal Adjust Accumulator instruction.

GENERAL-PURPOSE REGISTERS

The Z-80 registers consist of fourteen general-purpose 8-bit registers designated A, B, C, D, E, H, and L and A′, B′, C′, D′, E′, H′, and L′. Only one set of seven registers and the corresponding flag register F or F′ can be active at any given time. A special Z-80 instruction selects A and F or A′ and F′, while a second instruction selects B, C, D, E, H, L, or B′, C′, D′, E′, H′, or L′. The possible com-

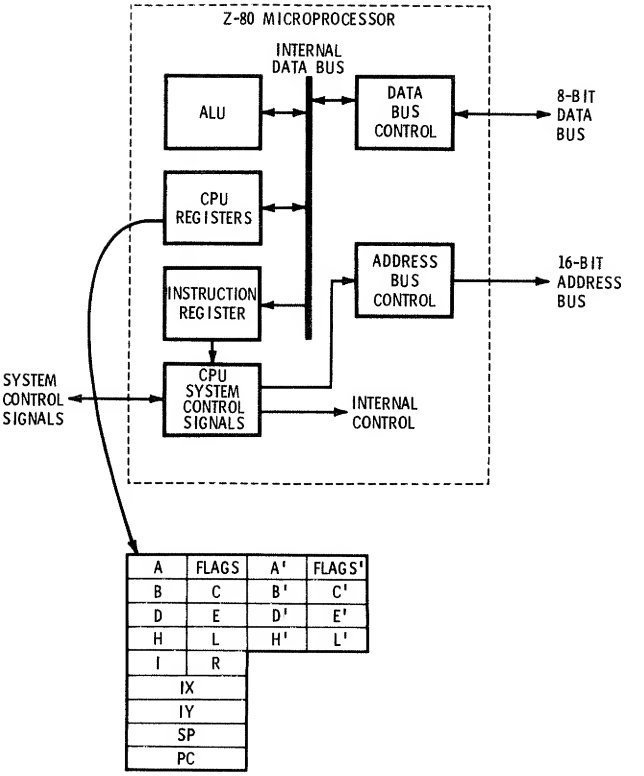


Fig. 2-1. Z-80 Microprocessor architecture.

binations of A and F and the remaining six general-purpose registers are shown in Fig. 2-2.

The advantage in two blocks of general-purpose registers is that a programmer may rapidly switch from one block to another. In the simplest case, this provides more register storage in the CPU. Register storage in the CPU is to be preferred over storage in memory as data can be accessed by a program much more rapidly from CPU registers than from external memory. In a more sophisticated use of the block switching capability, the unused set of registers may be used to hold the *environment* after receiving an *interrupt*. This concept will be discussed in a later chapter in this section.

Just as in the 8080, the general-purpose registers are somewhat specialized in function. Eight bits of data may be moved between memory and any of the seven registers or from one register to the next. Arithmetic and logical operations, however, such as adding two operands or exclusive ORing two operands can only be done using the A register (or A') and another register or memory location.

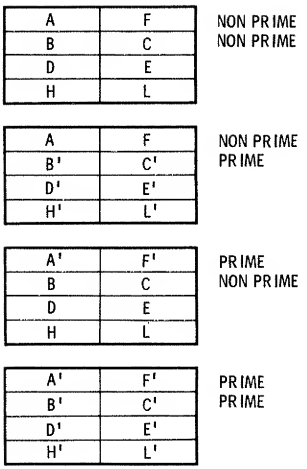


Fig. 2-2. Register block combinations.

The result of the operation always goes into the A register. In general, then, the currently selected A register is the main register for performing arithmetic and logical operations as shown in Fig. 2-3.

The remaining six registers are grouped into *register pairs* B,C; D,E; and H,L. For many operations in the 8008, 8080, and Z-80 the data within the three register pairs represents a memory address. The H,L registers, for example, originally specified a *H*igh memory address of eight more significant bits and a *L*ow memory address of eight less significant bits as shown in Fig. 2-4. The same is true of the B,C and D,E registers. In the 8080, the capability also was provided to allow the B,C and D,E to specify a memory address, giving three register pairs that could hold a memory address *pointer* to data in memory. In general, the three register pairs will hold memory addresses as shown in Fig. 2-4, although a second use for them is to allow *double-precision arithmetic.*

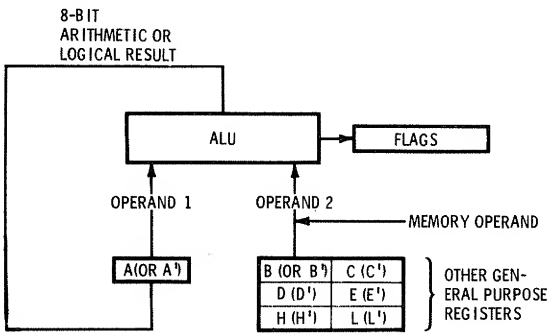


Fig. 2-3. Arithmetic and logical operations.

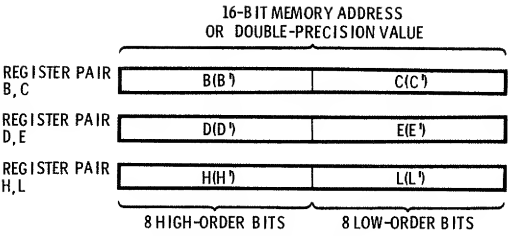


Fig. 2-4. Register pairs.

Double-precision arithmetic involves adding, subtracting, incrementing (adding one), or decrementing (subtracting one) a 16-bit value. Most arithmetic and logical operations in the Z-80 are oriented towards 8-bit operations, but the Z-80 allows limited operations between the register pairs and the stack pointer and index registers IX and IY. The general philosophy for this probably evolved from the requirement to manipulate memory address pointers in some convenient fashion, since all external memory addresses are 16-bit addresses and two 8-bit operations would have to be performed if 16-bit arithmetic were not implemented. Fig. 2-5 shows the use of the register pairs in double-precision operations.

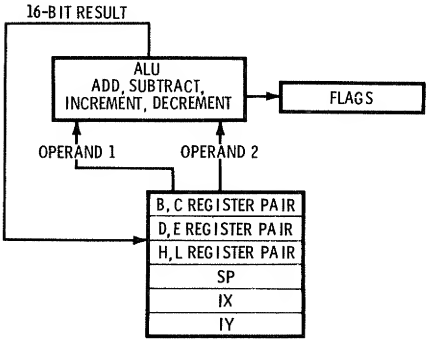


Fig. 2-5. Register pair double-precision operation.

FLAG REGISTER

The flag register is selected along with the A register. At any given time A and F or A′ and F′ are selected. Although the *flag* register is a register of eight bits as are the other seven CPU registers, it is more a collection of eight bits conveniently grouped into one register than a general-purpose register. The bits within the flag register specify various CPU conditions that have occurred after an arithmetic, logical, or other CPU operation. For example, it is convenient to know if the result of the addition of two operands resulted in a zero result,

a positive (zero or greater) result, or a negative result. A zero flag and a sign flag in the flag register may be tested by the program after the add to determine the nature of the result. Other flags are the carry flag (C), the carry from the high order bit of the accumulator, the parity/overflow flag (P/V), specifying a parity or overflow condition, the half carry flag (H), which is essentially a bcd carry or borrow from the low order bcd digit, and the subtract flag (N), set for bcd subtract operations. The flag register format is shown in Fig. 2-6. The interaction of CPU operations and the flags is discussed in

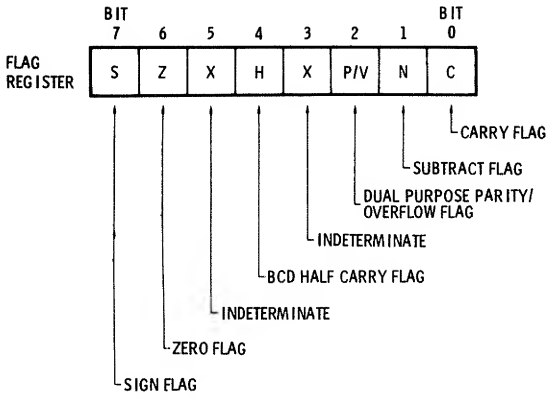


Fig. 2-6. Flag register format.

detail in a later chapter in this section. Throughout this book the term flags, flag register, and condition codes will be used interchangeably.

SPECIAL-PURPOSE REGISTERS

The remaining CPU registers that are available to the programmer are the I, R, IX, IY, SP, and PC registers. Two of these registers are exactly the same as they are in the 8080, the SP, or Stack *P*ointer, and PC, or *P*rogram *C*ounter. The PC register is a 16-bit register that holds the location of the current instruction being fetched from memory. Instructions in the Z-80 are one, two, three, or four bytes long. If a sequence of eight instructions is being executed, as shown in Fig. 2-7, the PC will hold the indicated values. Note that the PC always points to the start of the next instruction, and that the CPU will automatically increment the PC by one, two, three, or four depending on the length of the instruction being executed. The PC is available to the programmer only in the sense that it may be *loaded* or *stored*. No arithmetic or logical operations on the PC are permitted.

Whereas the PC contains a pointer to external memory that specifies the address of the next instruction to be executed, the SP contains a pointer to an external memory stack. The concept of a memory stack is not unique to microprocessors, but virtually every microprocessor does have stack capability. The external memory stack is simply an area of memory set aside for temporary storage of CPU registers, the flag register, and the program counter. Certain instructions cause transfer of control from the current *jump* or *branch* in-

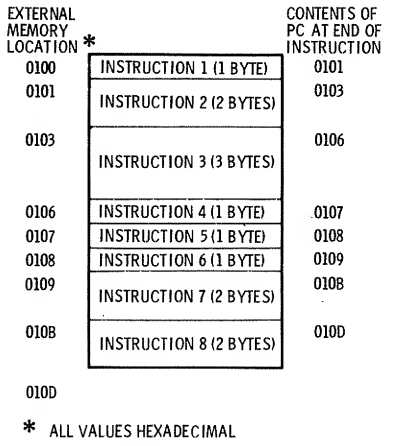


Fig. 2-7. Program counter operation.

struction to another instruction and cause the current contents of the program counter (pointing to the instruction after the jump or branch) to be automatically saved in the stack area. This saves the location so that at some later time a return may be made back to the next instruction in sequence after the jump or branch.

Not only is the PC saved for certain types of jumps or branches, but it is automatically saved for *interrupts*. Here, the address of the current instruction being executed is saved in the stack as the interrupt occurs and a special interrupt processing routine is entered. This action will be discussed in detail in a later chapter in this section. Lastly, CPU registers and the flag register may be saved and retrieved from the stack under program control using special stack instructions.

As data is entered or *pushed* into the stack area, the stack pointer is decremented by one count. As data is retrieved from the stack or *pulled*, the stack pointer is incremented by one count. A good analogy to stack operation is a poker hand that is laid down on the table in a pile consisting of King of Hearts, Jack of Spades, and Ace of Diamonds with the King at the bottom. When the cards are retrieved, the first card picked up is the last laid down, the Ace of Diamonds, followed by Jack of Spades and King of Hearts. This type

of stack operation is a LIFO operation, or *last in, first out*. The contents of the SP during a typical instruction sequence is shown in Fig. 2-8. Note that the stack builds from higher numbered memory to lower numbered memory as more data is stored in the stack.

The remaining registers of the Z-80 are not contained in the 8080. The index registers IX and IY are two 16-bit registers that permit indexed addressing in Z-80 programs. While the 8080 had indexed-like instructions, it did not permit true indexing. When an instruction is executed in an indexed addressing mode, one of the two index registers is used to calculate the memory address of the operand.

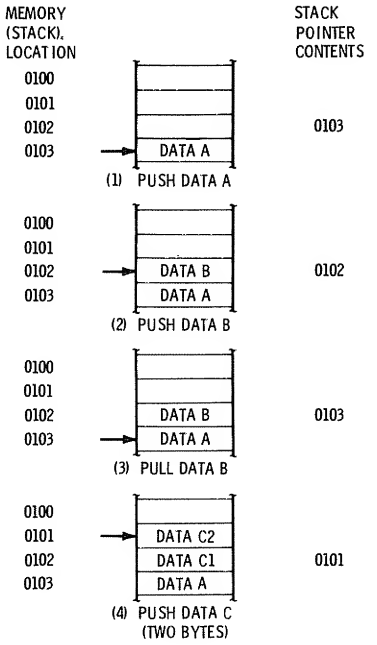


Fig. 2-8. Stack Pointer (SP) operation.

The *effective address* of the memory operand is obtained by adding the contents of the index register and a 16-bit value contained in the *displacement field* of the instruction employing the indexed addressing mode. Indexed operations of this kind are extremely powerful for efficient programming and will be discussed in more detail later.

The Interrupt Vector Register I is an 8-bit register that can be loaded with 8 bits of data specifying a memory address. This address, when combined with a lower-order 8 bits of address supplied by the interrupting device, represent a memory address whose contents in turn specify the memory address of the software *interrupt handling routine* for the device. Suppose that a paper-tape reader

interrupts the Z-80. After the Z-80 recognizes the interrupt, it signals the paper-tape–reader *controller* to pass over the low order 8 bits of the address. The paper-tape–reader controller then passes over the 8 least significant bits of the address which are combined with the 8 higher order bits of the I register. If the paper-tape reader supplied 14H (A suffix of "H" will represent base 16, or hexadecimal in all subsequent discussions) and the I register contained FFH, then the combined address would represent FF14H. The Z-80 control logic would then go to external memory location FF14H, pick up *its* contents and transfer control to the location specified, in this case E000H as shown in Fig. 2-9. In general, the I register holds the 8 most significant bits of an interrupt *vector* table which may hold interrupt vectors for 128 interrupting devices.

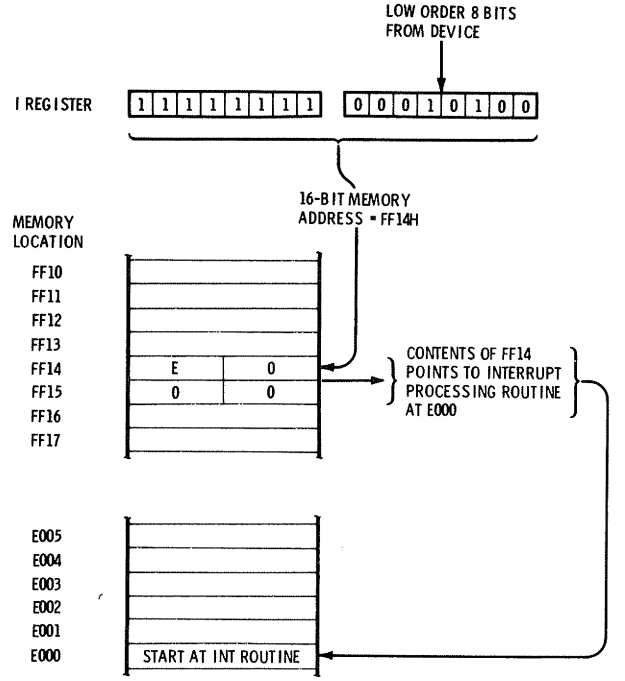


Fig. 2-9. I Register actions.

The I register is used in one of three interrupt modes which the Z-80 may utilize under program control. One of the other two modes is identical to the 8080 interrupt action, allowing up to eight vectored interrupts. The last interrupt mode permits a special ninth interrupt. In addition to the three external interrupt modes, a *nonmaskable* (always active) external interrupt permits a high-priority

interrupt to yet another interrupt location. All four kinds of interrupt groupings are discussed in a later chapter in this section.

The last special-purpose register is the 7-bit Memory Refresh register R. When external memory is made up of *dynamic* memories, the R register allows automatic refreshing of this kind of semiconductor memory which periodically (typically every 2 milliseconds) needs to have every cell read or refreshed to retain its contents. The contents of the R register are incremented by one after every instruction *fetch* and the contents are sent out along the least significant 7 bits of the address bus while the Z-80 CPU is not accessing memory. Every cell of external memory with a predefined configuration of its address bits equal to the R register can now be refreshed without fear of contention (simultaneous read) of the same memory cell by the Z-80 CPU. The R register is normally not used by the programmer.

MICROCOMPUTER COMPONENT PARTS

As in any microcomputer, the microprocessor chip itself does not constitute the complete computer system. Fig. 2-10 shows the component parts of a typical Z-80 system. The Z-80 microprocessor chip

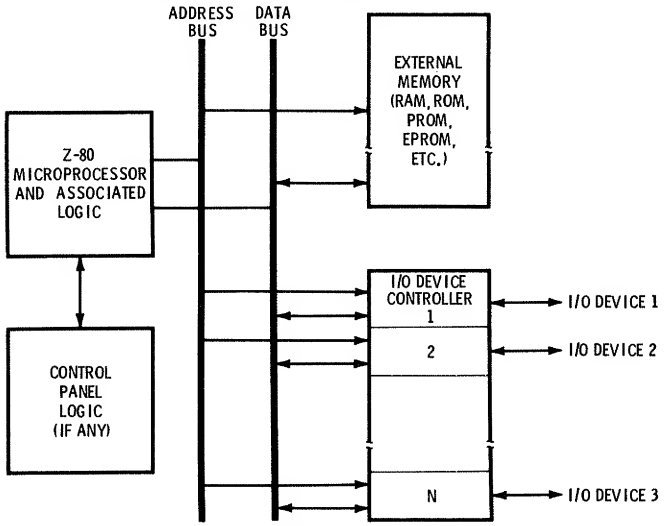


Fig. 2-10. Z-80 Microcomputer system component parts.

along with supporting circuitry interfaces to external memory. Control signals are passed between CPU circuitry and external memory, memory addresses are passed along the 16-bit address bus, and data is passed along the 8-bit address bus. External memory may be any

combination of the many kinds of external memory available today. RAM (*r*andom *a*ccess *m*emory) is semiconductor memory that can be both read and written into. ROM (*r*ead *o*nly *m*emory) is a production-type memory that contains a program or data or both which can be read but not altered. PROM (*p*rogrammable *r*ead *o*nly *m*emory) may be *programmed* in the field with inexpensive equipment, but may not be altered once programmed. EPROM (*e*rasable *p*rogrammable *r*ead *o*nly *m*emory) may be programmed for a read only operation, but may be periodically erased under ultraviolet light. Many wags have suggested another type, a WOM or *w*rite *o*nly *m*emory, but in most cases the former memory types are commonly used.

The Z-80 microprocessor and associated CPU circuitry interface to I/O *device controllers* along with external memory. I/O device controllers perform several functions. Firstly, the I/O device controllers *buffer* data passing between the Z-80 CPU registers or external memory and the I/O device. The buffering matches the high-speed data-transfer rate of the Z-80 CPU to the relatively low-speed rate of the I/O device. It is important for the CPU not to have to wait until the I/O device accepts data, as the wait time may represent tens of thousands of Z-80 instructions. A Teletype Corporation ASR-33 Teletype, for example, accepts data at the rate of 10 bytes per second. While waiting for the Teletype to accept a byte of data, the Z-80 microprocessor could be executing 1/10 second worth of instructions or about 30,000 instructions. The Teletype controller allows the Z-80 to pass a byte in several microseconds and signals the Z-80 when the Teletype is done processing the data from the Teletype device controller.

Another function performed by the I/O device controller is formatting of the data. A floppy disc transmits data as a *serial* bit stream. The floppy disc controller, among other functions, converts the serial bit stream into 8-bit *parallel* bytes in proper format for transmission to the Z-80 CPU over the data bus.

A third function of the I/O device controller is that of *level conversion.* Data from CPU logic is in TTL (or Transistor-Transistor Logic) signal levels, which are nominally 0 volts and 5 volts. A Selectric I/O typewriter may require 24 to 48 volts to drive the solenoids of the Teletypes and obviously some voltage level conversion is required.

Other functions of the I/O device controller are timing, synchronization, control-signal *handshaking,* and transmission of device status. A wide range of I/O devices interface to the Z-80 through their respective device controllers, ranging from 5 character-per-second Teletype equipment, audio cassette equipment, analog-to-digital converters, and 100,000 byte-per-second graphic display

equipment, to mention a few of the virtually dozens of devices. Some of the more common generic types will be covered in a later chapter of this section along with special-purpose LSI chips of the Zilog Z-80 family which are designed to permit ease of interfacing.

The last functional block of Fig. 2-10 is that of the control panel. Many current microcomputers have dispensed with a control panel except for one sparsely configured with a power switch and a reset switch. Pressing the reset switch causes a nonmaskable interrupt which transfers control to a special *monitor* program in PROM or ROM memory. The monitor program allows the user to interrogate memory locations, change the contents of memory locations, modify registers, load and save programs on I/O devices and other functions. If a control panel is present, it performs the same functions as the monitor program by allowing the user to manually address, examine, and change data in CPU registers and memory. The only advantage that a control panel would have over a monitor program is that only the CPU, memory, and control panel are required to execute programs. However, any viable system must have some kind of I/O device and in almost all cases, the control panel is an added complexity.

Section III discusses many of the more popular Z-80 microcomputer systems and will give the reader an overview of what is available in current Z-80 microcomputers insofar as system architecture is concerned.

CHAPTER 3

Interface Signals and Timing

The Z-80 CPU chip is a 40-pin dual in-line package. The pinout of the chip is illustrated in Fig. 3-1, with the pins logically grouped according to function, rather than the actual physical representation.

ADDRESS AND DATA BUS

The address bus is represented by signals A15 through A0, where A15 is the most significant bit of the address bus and A0 is the least significant bit. A15 through A0 are active high and are a tri-state output meaning that when the address bus is inactive, its outputs are in a high-impedance state. The address bus lines considered together represent a 16-bit memory or device address. Since 2^{16} addresses can be held in 16 bits, external memory of 65536_{10} or 64K may be addressed directly by the Z-80 CPU. When I/O devices are addressed, the least significant eight lines of the address bus, A7-A0, hold the I/O device address, which may be 0 through 255_{10}. In addition to memory or I/O device addresses, the least significant seven lines of the address bus hold the contents of the R, or Memory Refresh Register, for certain times during execution of each instruction.

The data bus, signals D7 through D0, are tri-state active high signals with D7 representing the most significant bit and D0 representing the last significant bit. The data bus is bidirectional, permitting data to be transferred to CPU registers from external memory or I/O devices or from CPU registers to external memory or I/O.

BUS CONTROL SIGNALS

Associated with the address bus and data bus are two CPU bus control signals, the input signal $\overline{\text{BUSRQ}}$ and the output (acknowl-

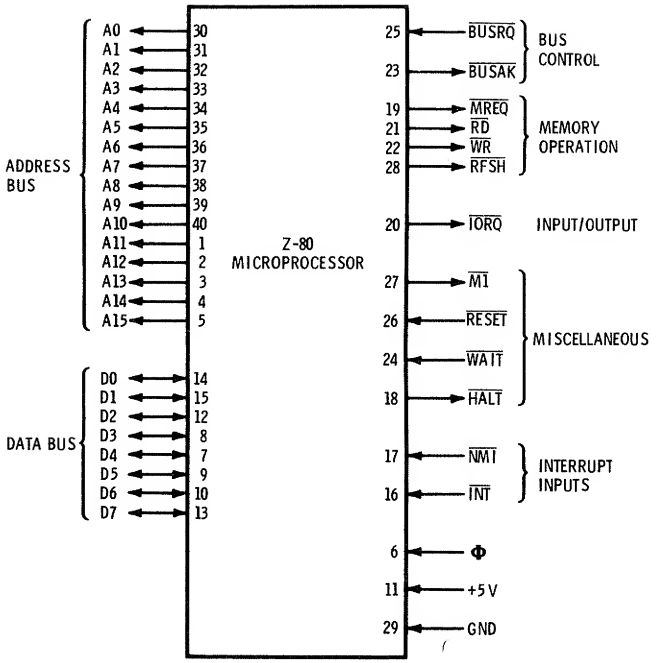


Fig. 3-1. Z-80 interface signals.

edge) signal $\overline{\text{BUSAK}}$. Signal $\overline{\text{BUSRQ}}$ is an active low signal that is generated by an external device to gain control of the CPU busses. During the time the external device has control of the busses, it will probably perform a direct-memory access (DMA) operation. DMA permits an external device to go directly to memory and transfer data between memory and the device. The CPU must be "locked out" during a DMA operation to avoid the conflict of the CPU requesting memory service at the same time and from the same memory location as an external device. When the external device *brings down* (logic 0) the $\overline{\text{BUSRQ}}$, Bus Request signal, the CPU responds with acknowledge signal $\overline{\text{BUSAK}}$, Bus Acknowledge. $\overline{\text{BUSAK}}$ is an active low output that signifies that the address bus, data bus, and CPU output-control signals are now in the high-impedance state and can be controlled by an external device for DMA operations.

MEMORY SIGNALS

There are four signals associated with memory operation, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, and $\overline{\text{RFSH}}$. The first, $\overline{\text{MREQ}}$, Memory Request, is a tri-state active low signal indicating that the address bus holds a valid

memory address. Essentially, this is part of a *chip enable* signal for external memory to inform external memory to output data for a memory read or to input data for a memory write. The $\overline{RD}$ and $\overline{WR}$ signals are tri-state active low outputs to external memory indicating whether the memory operation is to be a read or write. When signal $\overline{MREQ}$ goes low, either $\overline{RD}$ or $\overline{WR}$ will also be low during a portion of the machine *cycle*. When $\overline{MREQ}$ and $\overline{RD}$ are both low, an external memory read will be performed. When $\overline{MREQ}$ and $\overline{WR}$ are both low, an external memory write will be performed. Both reads and writes utilize the address on the address bus and transfer data along the data bus.

The $\overline{RFSH}$ signal is not associated with normal memory operation. It is used only when dynamic memories are used as external memories. Dynamic memories periodically require a *refresh* to maintain the data stored within the memory cell. This is essentially a memory read operation with the data not being transferred from the memory. Typical dynamic memories are set up so that a refresh signal can be input to the memory, along with five or six address line inputs. To refresh an entire memory, six address line inputs would require sixty-four separate refreshes (2^6) with the entire refresh cycle lasting no longer than 2 milliseconds. When the output signal $\overline{RFSH}$ is low and signal $\overline{MREQ}$ is also low, external dynamic memory will use the contents of the least significant seven bits of the address bus to implement one of the refresh cycles. $\overline{RFSH}$ is active at every instruction fetch, and since the R register is continually being incremented after each fetch, the address lines will continually reflect a new address for the next refresh cycle. For the above example of six address line inputs, it will take sixty-four instruction cycles to refresh dynamic memory or approximately 256 microseconds (.256 milliseconds) at about 4 microseconds per instruction, average.

INPUT/OUTPUT SIGNALS

Signal $\overline{IORQ}$ is a tri-state, active low output signal used for *I*nput/*O*utput *R*equests. When signal $\overline{IORQ}$ goes low, the least significant eight bits of the address bus, A7-A0, hold an I/O device address. Signals $\overline{RD}$ and $\overline{WR}$ must then be used to determine whether the I/O operation is to be an I/O read or write. Signal $\overline{IORQ}$ is also used in conjunction with signal $\overline{M1}$ for interrupt responses as discussed below.

OTHER CPU SIGNALS

Signal $\overline{M1}$ is an active low output signal that indicates the microprocessor is in the fetch cycle of the instruction. Every instruction

has a fetch cycle as the first byte of the instruction, the operation code, is fetched from memory and then decoded. In the Z-80, unlike the 8080, several instructions have two-byte operation codes and signal $\overline{M1}$ will be low during each of the fetches of one byte.

The $\overline{RESET}$ signal is an active low input signal that is used as a master CPU reset. This signal would be brought low immediately after power up, or at any time when the microcomputer system was to be reset. When $\overline{RESET}$ is brought low, the following actions occur:

1. The interrupt enable flip-flop is disabled, preventing system interrupts except for $\overline{NMI}$ (see below).
2. Register I, the Interrupt Vector Register, is set to 00H.
3. Register R, the Refresh Register, is set to 00H.
4. Interrupt mode 0 is set.
5. The address bus goes to a high-impedance state.
6. The data bus goes to a high-impedance state.
7. All output-control signals go to the inactive state.

The $\overline{WAIT}$ signal is a signal associated with slow memories or I/O devices. As long as the $\overline{WAIT}$ signal is low, the CPU will "mark time," doing nothing, while the external memory or I/O device responds to a previous memory or I/O request. The $\overline{WAIT}$ signal enables slow memories or (rarely) slow I/O devices to be interfaced to the Z-80 without buffering.

The $\overline{HALT}$ signal is an active low output signal that goes low during the time that a $\overline{HALT}$ instruction is being executed. A $\overline{HALT}$ instruction in a program is typically used for one of two conditions. Either the program has performed all of its functions and terminated, or a halt has been reached and the program is waiting for an interrupt to occur. When the CPU is in a halt state, it performs no-operations instructions (NOP) to ensure proper memory refresh activity.

INTERRUPT-RELATED SIGNALS

The remaining logic signals are associated with interrupt processing. Signal $\overline{NMI}$ is a negative-edge triggered input that specifies a *non*maskable *i*nterrupt is to be performed. When this signal is momentarily brought low, the CPU will recognize this interrupt at the end of the current instruction. When the CPU recognizes the NMI interrupt, the following actions occur:

1. The current contents of the program counter PC is saved in the memory stack.

2. The CPU transfers control to memory location 0066H, that is, instruction execution starts from location 0066H which must contain an NMI interrupt processing program.

An NMI interrupt of this kind cannot be disabled and will *always* be recognized by the CPU at the end of the current instruction cycle. The exceptions to this are that signal $\overline{\text{BUSRQ}}$ will take precedence over a $\overline{\text{NMI}}$ signal, and that a continuous $\overline{\text{WAIT}}$ state will prevent the current instruction from ending and thus prevent the NMI from being recognized.

The main interrupt request is signal $\overline{\text{INT}}$, an active low input signal that is supplied by external devices to cause an interrupt. The $\overline{\text{INT}}$ signal will be recognized by the CPU at the end of the current instruction if the interrupt enable flip-flop IFF in the CPU has been set by the program and if the $\overline{\text{BUSRQ}}$ signal is not active. If these conditions are met, the CPU accepts the interrupt and acknowledges the interrupt by sending out an $\overline{\text{IORQ}}$ during the fetch ($\overline{\text{M1}}$) time of the next instruction. Since $\overline{\text{IORQ}}$ *never* occurs during $\overline{\text{M1}}$ for an I/O instruction, the interrupting device recognizes the $\overline{\text{IORQ}}$ and $\overline{\text{M1}}$ condition as an interrupt acknowledge. Further actions taken for this interrupt are discussed later in this section.

CPU ELECTRICAL SPECIFICATIONS

The electrical specifications for the Z-80 microprocessor chip are shown in Chart 3-1. All inputs and outputs are TTL compatible facilitating interfacing. There is only one power-supply voltage, a 5-volt power supply. The Z-80 microprocessor chip alone requires a maximum current of 200 milliamps. Unlike the 8080, there is only a single-phase clock input required, which is also at TTL levels. The frequency of the clock for the original Z-80 was 2.5 megahertz, however, faster versions will accept a 4-megahertz clock at this time of writing. Detailed specifications for other dynamic parameters are provided in Appendix A.

CPU TIMING

All instruction execution in the Z-80 may be broken down into a set of basic cycles. There are two kinds of cycles, the most basic being a clock cycle, or T cycle. If a 4-MHz clock is being used for the Z-80, each T cycle will be a constant length (period) of 250 nanoseconds as shown in Fig. 3-2. The T cycles are used to control operations within a larger cycle called the machine cycle, or M cycle. Every instruction executed within the Z-80 consists of from one to six machine cycles (with the exception of special block-

Chart 3-1. Z-80 Electrical Specifications

ABSOLUTE MAXIMUM RATINGS

Temperature Under Bias	0°C to 70°C
Storage Temperature	−65°C to +150°C
Voltage On Any Pin with Respect to Ground	−0.3V to +7V
Power Dissipation	1.1W

*Comment

Stresses above those listed under "Absolute Maximum Rating" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other condition above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

• DC CHARACTERISTICS

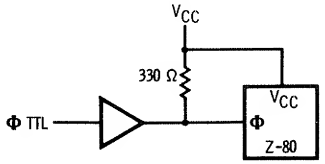
$T_A = 0°C$ to 70°C, $V_{cc} = 5V \pm 5\%$ unless otherwise specified

Symbol	Parameter	Min.	Typ.	Max.	Unit	Test Condition
V_{ILC}	Clock Input Low Voltage	−0.3		0.45	V	
V_{IHC}	Clock Input High Voltage	V_{cc}[1]		V_{cc}	V	
V_{IL}	Input Low Voltage	−0.3		0.8	V	
V_{IH}	Input High Votlage	2.0		V_{cc}	V	
V_{OL}	Output Low Voltage			0.4	V	$I_{OL} = 1.8$ mA
V_{OH}	Output High Voltage	2.4			V	$I_{OH} = -100\ \mu A$
I_{CC}	Power Supply Current			200	mA	$t_c = 400$ nsec
I_{LI}	Input Leakage Current			10	μA	$V_{IN} = 0$ to V_{cc}
I_{LOH}	Tri-State Output Leakage Current in Float			10	μA	$V_{OUT} = 2.4$ to V_{cc}
I_{LOL}	Tri-State Output Leakage Current in Float			−10	μA	$V_{OUT} = 0.4$ V
I_{LD}	Data Bus Leakage Current in Input Mode			±10	μA	$0 \leqslant V_{IN} \leqslant V_{cc}$

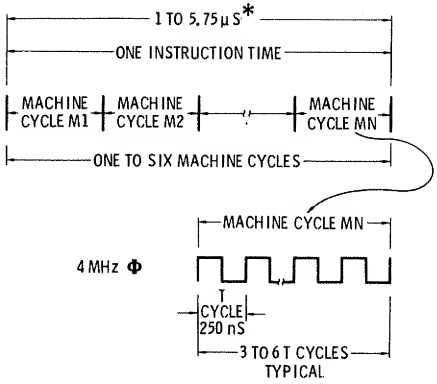
• CAPACITANCE $T_A = 25°C$, f = 1 MHz

Symbol	Parameter	Typ.	Max.	Unit	Test Condition
C_Φ	Clock Capacitance		20	pF	Unmeasured Pins Returned to Ground
C_{IN}	Input Capacitance		5	pF	
C_{OUT}	Output Capacitance		10	pF	

[1] Clock Driver



An external clock pull-up resistor of (330Ω) will meet both the ac and dc clock requirements.



* MOST INSTRUCTIONS, 4-MHz CLOCK

Fig. 3-2. Basic instruction cycles.

related instructions), and each of the machine cycles is comprised of three to six T cycles as shown in the figure.

There are seven basic machine cycles that can occur during Z-80 operation:

1. Operation code fetch cycle (M1 cycle)
2. Memory data read or write cycle
3. I/O read and write cycles
4. Bus Request/Acknowledge cycle
5. Interrupt Request/Acknowledge cycle
6. Nonmaskable Interrupt Request/Acknowledge cycle
7. Exit from a HALT instruction

M1 CYCLE

Every instruction execution is made up of one operation code fetch cycle, or M1 cycle. A few instructions have two bytes for the operation code and therefore have two M1 cycles. An M1 cycle allows the CPU to read the operation code byte from external memory, decode the operation to be performed, and implement a portion or possibly all of the operation (for short instructions that are one machine cycle long.) Fig. 3-3 shows the timing diagram for an INC R instruction which will also illustrate the M1 cycle. The INC R takes only one machine cycle to fully execute the M1 cycle. Four T cycles are required.

As the CPU enters the M1 cycle, signal M1 falls to indicate that this cycle is active. The contents of the program counter is gated to the address bus in preparation for the fetch of the op code of the next instruction. On the falling edge of T1 signals $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$

go low, indicating to the external memory that there is a valid memory address on the address bus. The external memory will now gate the contents of the specified memory location onto the data bus somewhere before the rising edge of T3 (unless it is a slow memory as discussed later in this chapter). On the rising edge of T3, the operation code byte on the data bus is clocked into the CPU. Shortly thereafter, the $\overline{RD}$ signal goes to an inactive level, along with $\overline{MREQ}$ and $\overline{M1}$. The remaining two T cycles of $\overline{M1}$ are used to provide a refresh time for external dynamic memories. Signal $\overline{RFSH}$ is brought low and $\overline{MREQ}$ is again active to indicate to external dynamic memory that refresh can proceed. The data bus will now have the contents of the R register present to provide a refresh address.

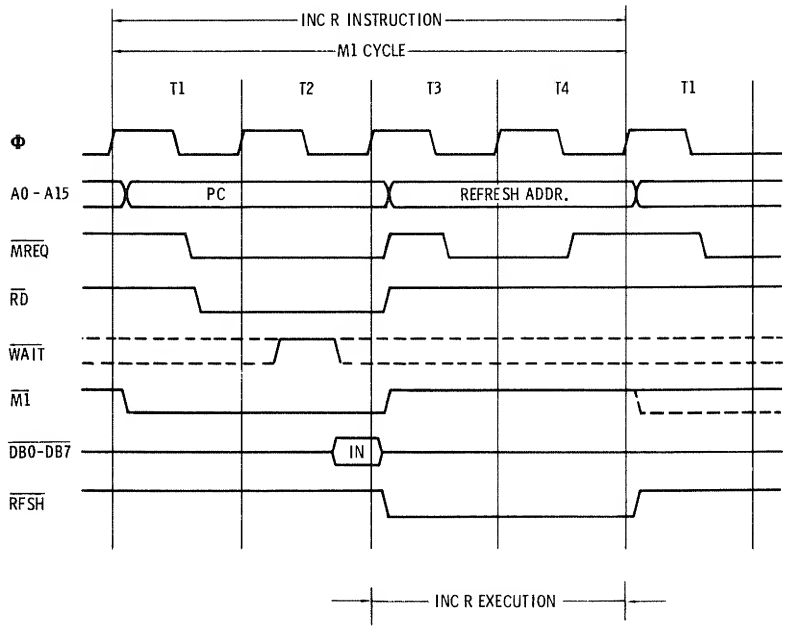


Fig. 3-3. M1 (op code fetch) cycle.

During the last two T cycles of M1 the CPU decodes the operation code of the instruction, which is an INC R. The INC R takes the contents of the specified general-purpose register R (A, B, C, D, E, H, or L or their primes), increments it by one count, and puts the result back into the register, setting the appropriate condition codes. Since no further memory accesses have to be made and the accesses of CPU registers can easily be made in several hundred nanoseconds, no further machine cycles are required.

MEMORY DATA READ AND WRITE CYCLES

The memory read and write cycles will be illustrated with examples of the execution of two instructions. Fig. 3-4 shows the execution of an LD R, (HL) instruction which loads the contents of the memory location pointed to by the H,L register pair into CPU register R. The M1 cycle is identical to that previously discussed. At the end of M1, the CPU has decoded the instruction and initiates a memory read cycle to obtain the eight-bit operand from memory. The address bus, $\overline{MREQ}$, and $\overline{RD}$ signals are activated just as in the case of the M1 cycle. The address bus holds the contents of the H,L register pair during this time and external memory gates the operand onto the data bus. On the falling edge of T3, the memory operand is clocked into the CPU, loading register R.

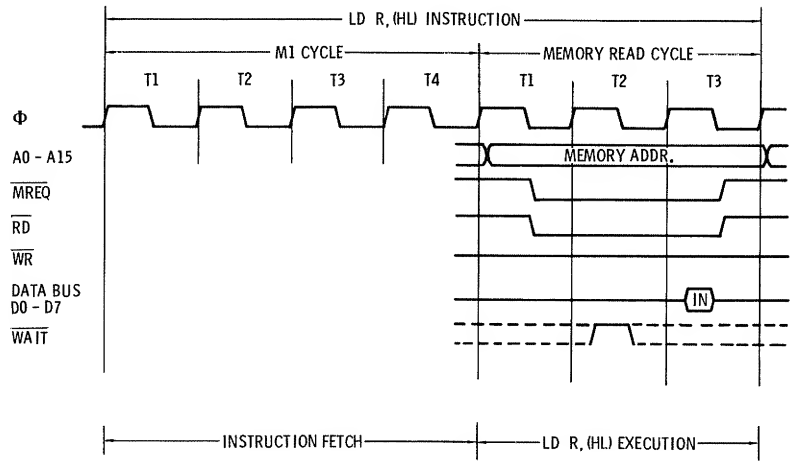


Fig. 3-4. Read cycle.

A memory write is shown in Fig. 3-5. The instruction in this case is an LD (HL), R which takes the contents of the specified CPU register R and writes it into the external memory location pointed to by the H,L register. The $\overline{MREQ}$ and address bus outputs are active as in the previous examples. No $\overline{RD}$ signal is output, but the contents of the specified CPU register are gated onto the data bus after the falling edge of T1. This data remains on the data bus and at the falling edge of T2 the $\overline{WR}$ signal becomes active. With $\overline{MREQ}$ and $\overline{WR}$ active, external memory writes the data on the data bus into the specified memory location, using address bus outputs A15-A0.

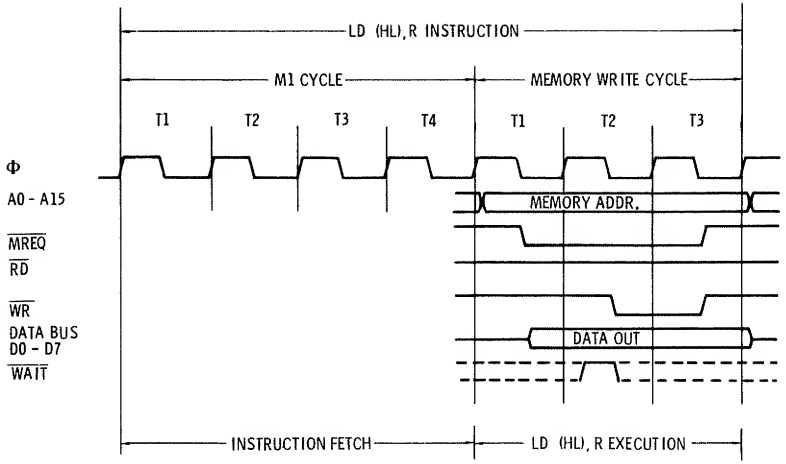


Fig. 3-5. Write cycle.

I/O READ AND WRITE CYCLES

An I/O Read or Write cycle occurs during an input or output instruction. Input and output instructions generally are three or four machine cycles long and from 10 to 20 T cycles (2.5 to 5 microseconds long for a 4-MHz clock). The more sophisticated I/O block-transfer instructions (INIR, INDR, OTIR, OTDR) transfer up to 256 bytes, however, and repeat machine cycles until all bytes have been transferred, resulting in total instruction times that are dependent on the number of bytes to be transferred and the speed of the I/O device. Fig. 3-6 shows an input cycle and Fig. 3-7 shows

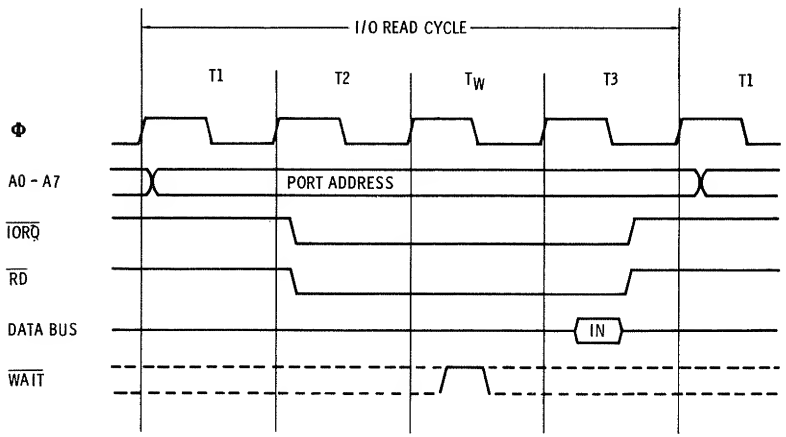


Fig. 3-6. I/O Read cycle.

an output cycle. The I/O device address is placed on lines A7-A0 of the address bus at the start of the machine cycle and the $\overline{IORQ}$ is enabled after the rising edge of T2. If a read is taking place, signal $\overline{RD}$ is enabled at the same time as $\overline{IORQ}$. The external device controller recognizes a read by the $\overline{IORQ}$ and $\overline{RD}$ and gates its data onto the data bus, where, on the falling edge of T3, it is clocked into the CPU.

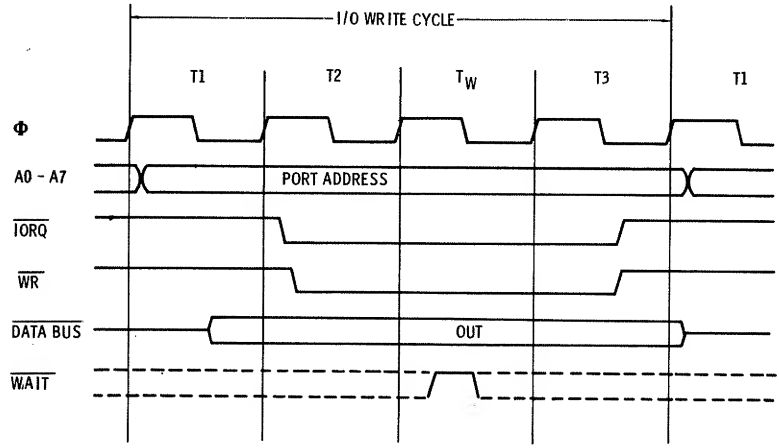


Fig. 3-7. I/O Write cycle.

If a write is taking place, the $\overline{WR}$ signal is enabled in place of the $\overline{RD}$ at the same time as $\overline{IORQ}$. Previous to the $\overline{WR}$ data from the CPU has been placed in the CPU register (during T1). This data is available during the remainder of the write cycle and the external I/O device controller will input it somewhere in this period.

Note that for both input and output cycles, signal $\overline{WAIT}$ is internally enabled after T2. This causes the CPU to defer further I/O processing until the $\overline{WAIT}$ line again is deactivated and effectively adds one clock cycle to the time of the input and output cycle. This condition is implemented to give the CPU additional time to sample the external $\overline{WAIT}$ line to respond to slow I/O devices. Additional $\overline{WAIT}$ states may be imposed by the external I/O device controller for as long as it takes the I/O device controller to execute the I/O instruction. These would be inserted for n number of T cycles after the CPU-imposed wait cycle.

BUS REQUEST/ACKNOWLEDGE CYCLE

At any time, an external device can gain control of the address bus A15-A0, data bus D7-D0, and $\overline{MREQ}$, $\overline{RD}$, $\overline{WR}$, $\overline{IORQ}$, and

$\overline{\text{RFSH}}$ lines by enabling the input signal $\overline{\text{BUSRQ}}$. Normally, the reason for this would be to allow an external device controller to communicate directly with external memory to transfer data between high-speed I/O devices and memory without CPU interference (Direct Memory Access or DMA). See Fig. 3-8. When signal $\overline{\text{BUSRQ}}$ is enabled, the CPU detects the signal during the rising edge of the last T cycle of a machine cycle. The T cycle is then completed and on the next T cycle the CPU responds to the request by output signal $\overline{\text{BUSAK}}$. At the same time, the address bus, data bus, and other signals are set to the tri-state high-impedance state. Now any changes to the lines will not be affected by the CPU nor will the CPU affect the state of the lines. When the I/O device controller has completed the DMA transfer (typically one byte), it will deactivate $\overline{\text{BUSRQ}}$. This condition will be detected by the CPU on the next rising edge of a T cycle and it will bring up or disable $\overline{\text{BUSAK}}$ on the next T cycle after that. The CPU will then continue processing from the point at which it gave control to the bus requestor.

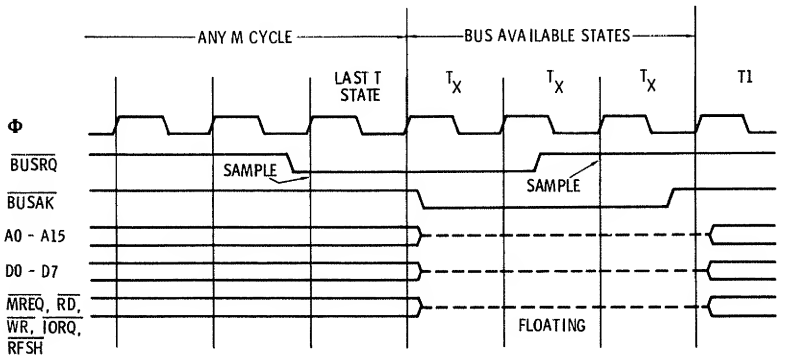


Fig. 3-8. Bus Request/Acknowledge cycle.

INTERRUPT REQUEST/ACKNOWLEDGE CYCLE

If the CPU interrupt enable flip-flop has been set to allow external interrupts, and if a bus request action is not taking place, the CPU is free to recognize external interrupts. An external device makes the interrupt request by enabling signal INT. During the rising edge of the last T cycle of the *last machine cycle* of an instruction, the CPU polls the state of the INT line, and, if low, starts an interrupt cycle as shown in Fig. 3-9. During T1 of the interrupt cycle, the M1 signal is enabled. T2 and two WAIT states are provided (the WAIT states are internally generated) to give sufficient time for external *daisy-chained* interrupt circuitry to respond to the

interrupt acknowledge and place an *interrupt response vector* on the data bus. The external interrupt logic identifies the interrupt acknowledge from the CPU by the combination of $\overline{M1}$ and $\overline{IORQ}$. After detecting these two signals, the external interrupt circuitry responds by placing the proper data on the data bus, which is clocked into the CPU during the rising edge of T3. During T3, the $\overline{M1}$ and $\overline{IORQ}$ signals are disabled and refresh action is started. Further action during the external interrupt cycle is dependent on the interrupt mode and is discussed later in this section.

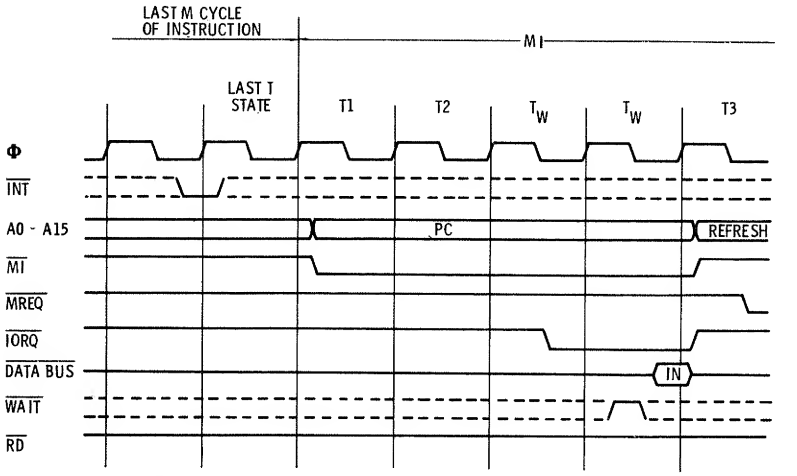


Fig. 3-9. Interrupt Request/Acknowledge cycle.

NONMASKABLE INTERRUPT REQUEST CYCLE

The CPU action during this machine cycle is shown in Fig. 3-10. The $\overline{NMI}$ signal cannot be disabled by the CPU interrupt enable

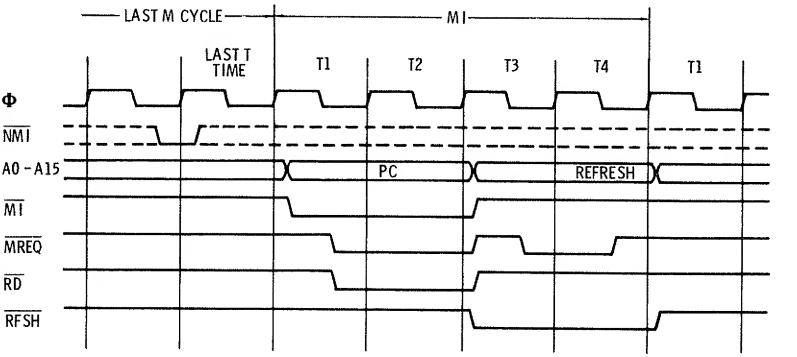


Fig. 3-10. Nonmaskable Interrupt Request cycle.

flip-flop. The NMI interrupt also takes priority over the external interrupt. It is recognized during the last T cycle of the last machine cycle of the current instruction as in the case of the external interrupt. Fig. 3-10 shows the first portion of this interrupt action. $\overline{IORQ}$ is not enabled since on external device needs to be notified that the interrupt was accepted. The first machine cycle is similar to a memory read operation, except that no data is read from external memory. Refresh operations are carried on in T3 and T4 as $\overline{RFSH}$ and $\overline{MREQ}$ are enabled, and the contents of the R register are placed on the address bus A7-A0. The NMI interrupt sequence is discussed later in this section.

EXIT FROM HALT INSTRUCTION

When a software HALT instruction is executed, signal $\overline{HALT}$ is enabled automatically by the CPU. The CPU continually generates M1 cycles for this HALT and does not advance the program counter. Data from memory is ignored. Refresh logic is enabled during the last two T cycles of M1 as before to enable proper refresh of external memory while the CPU is in the halted state. The HALT state can only be interrupted by a $\overline{RESET}$ or receipt of an $\overline{NMI}$

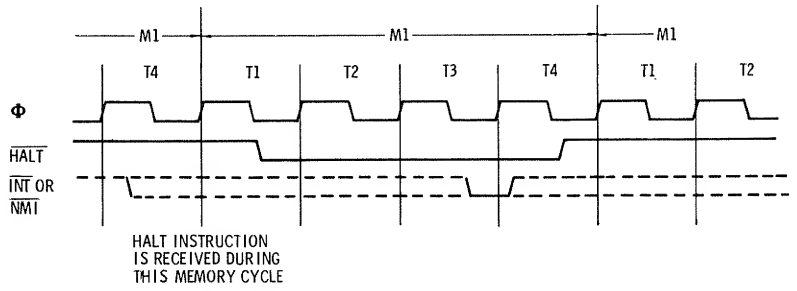


Fig. 3-11. Exit from HALT instruction.

or external interrupt, both of which cause normal interrupt processing as before and cause the CPU to advance the program counter to the next instruction before the program counter is stored in the memory stack. The HALT instruction exit is shown in Fig. 3-11.

MEMORY OR I/O WAIT STATES

In general, WAIT states may be initiated after any memory or I/O request. When external memory or I/O receives an $\overline{RD}$ or $\overline{WR}$ signal and an $\overline{MREQ}$ or $\overline{IORQ}$, it can respond by a $\overline{WAIT}$ input to the CPU. The CPU will detect the WAIT condition and defer

further processing until the memory or I/O device controller has had time to respond. External memories must be capable of responding in a little over one T cycle, or 250 nanoseconds for a 4-MHz clock, while input/output device controllers transferring data to the CPU have about two T cycles or 500 nanoseconds.

CHAPTER 4

Addressing Modes

The Z-80 has a wide repertoire of instructions, ranging from a simple instruction to set the interrupt enable flip-flop to a block-search instruction that searches a string of bytes for a given byte. Because of the wide range of functions that Z-80 instructions perform, instructions range in length from one byte to four bytes. In addition to differences in length, instructions differ in how external memory is addressed. Some instructions require no operand and can be executed during the last portion of an M1 (fetch) cycle. Other instructions require an operand from a CPU register and a second operand either from another CPU register or external memory. The second operand may be specified in a variety of ways. As an example, the ADD instruction adds two 8-bit operands. One of the operands is in the A register, while the second can be in another CPU register (Register Addressing), an immediate value in the ADD instruction itself (Immediate Addressing), in memory and pointed to by the contents of the HL register pair (Register Indirect Addressing), or in a memory location whose address is computed by adding a 16-bit displacement in the instruction and the contents of an index register (Indexed Addressing). This chapter will describe the various addressing modes of the Z-80, using examples of specific instructions. The next chapter discusses instruction types and describes which addressing modes are valid for each instruction.

The Z-80 has the following addressing modes, generally ordered from simple to complex:

1. Implied Addressing
2. Immediate Addressing

3. Extended Immediate Addressing
4. Register Addressing
5. Register Indirect Addressing
6. Extended Addressing
7. Modified Page Zero Addressing
8. Relative Addressing
9. Indexed Addressing
10. Bit Addressing

IMPLIED ADDRESSING

In this kind of addressing, the operation code of the instruction is fixed. There are no variable *fields* within the instruction, and the instruction always performs exactly the same function. Examples of this kind are the CPL and LD SP, IY instructions.

The format of the CPL, Complement Accumulator, is shown in Fig. 4-1. This instruction takes the contents of the A register, forms the ones complement (changes all zeros to ones and all ones to zeros) and stores the result back into the A register. No condition code bits are affected. The *source* and *location* are fixed and no other register can be used.

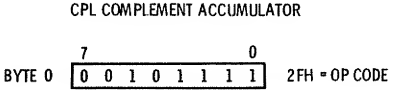


Fig. 4-1. Implied addressing in CPL instruction.

The format of the LD, SP, IY instruction is shown in Fig. 4-2. Load SP with IY takes the 16-bit contents of the IY register and transfers it to the SP register. The contents of the IY register remains unchanged and no condition-code bits are affected. The two-byte configuration FDF9H will always produce the same action of loading the SP register from the IY register.

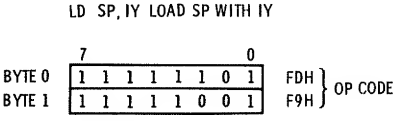


Fig. 4-2. Implied addressing in LD SP,IY instruction.

All of the instructions discussed in the next chapter under General-Purpose Arithmetic and CPU Control are of this kind, as are the instructions under the Exchange, Block Transfer, and Search Group. In the latter group, the actions are more elaborate, but the instruction format is fixed.

IMMEDIATE ADDRESSING

In the immediate addressing mode, the second or third byte of the instruction itself is the operand. Immediate addressing is valuable when it is necessary to load or perform an arithmetic or logical operation with constant data. The immediate addressing instructions ADD A,N and XOR N are examples of this addressing type.

The format of the ADD A,N instruction is shown in Fig. 4-3. The contents of the A register are added with the contents of the second

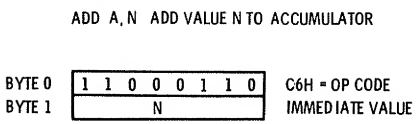


Fig. 4-3. Immediate addressing in ADD A,N instruction.

byte of the instruction and the result put into the A register. If two bytes of the ADD A,N instruction were C633H (ADD A,33H) and the A register contained 80H, 80H and 33H would be added to produce a result of B3H and this result would be put into the A register. The condition codes would also be set on the results of this instruction.

The format of the XOR N instruction is shown in Fig. 4-4. The contents of the A register are exclusive ORed with the second byte of the instruction and the result put into the A register. The condition codes are set on the result of the instruction. If the instruction were EE35H and the contents of the A register were 33H, 35H and 33H would be exclusive ORed to produce 06H, which would be put into the A register.

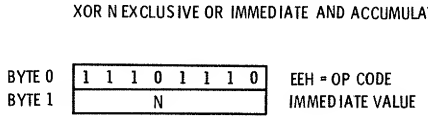


Fig. 4-4. Immediate addressing in XOR N instruction.

In general, the immediate addressing mode is used for instructions in the 8-bit Arithmetic and Logical Group discussed in the next chapter.

EXTENDED IMMEDIATE ADDRESSING

When the instruction is an immediate kind of instruction, but 16 bits of immediate data are required, the instruction format is of the "extended" immediate kind. The extended addressing mode is used in only a few instructions in the 16-Bit Load Group of in-

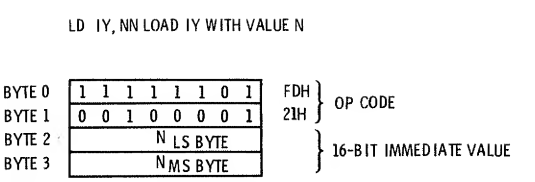


Fig. 4-5. Extended immediate addressing in LD IY,NN instruction.

structions. An example would be the instruction LD IX,NN which is shown in Fig. 4-5. Note that the first *two* bytes comprise the operation code, and that the next two are the immediate data itself. LD IX,NN loads the 16 bits of immediate data in bytes two and three of the instruction into the IX register. The condition-code bits are not affected. As in the case of all 8080 16-bit data, the data is grouped least significant byte followed by most significant byte. The instruction LD IX,123FH would load the IX register with 123FH and would appear as shown in Fig. 4-6.

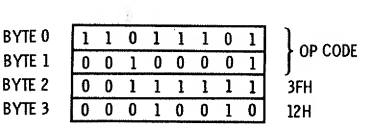


Fig. 4-6. Extended immediate addressing data arrangement.

REGISTER ADDRESSING

In the register addressing mode, one or more of the CPU registers is addressed by the instruction. The instruction format would contain a *field(s)* which would specify which CPU register(s) was to be utilized in performing the instruction. Examples of this kind of addressing would be the RL R and AND R instructions.

The RL R instruction format is shown in Fig. 4-7. The least significant 3 bits of word 1 of the 2-byte instruction is a 3-bit field that specifies one of the general-purpose CPU registers A, B, C, D, E, H, or L. This instruction takes the contents of register R and shifts it left one *bit position.* The most significant bit of the register is shifted into the carry, while the previous contents of the carry are shifted into the least significant bit position of the register. The condition-

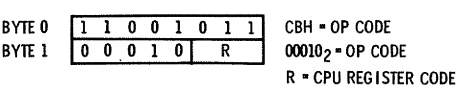


Fig. 4-7. Register addressing in RL R instruction.

code bits are set according to the results of the shift. Valid values for the R field of the instruction are as follows:

R	Register Shifted
000	B
001	C
010	D
011	E
100	H
101	L
111	A

Note that all bit permutations are possible except 110_2. If 110_2 were to be specified in this instruction, the instruction would become another kind of addressing mode, Register Indirect Addressing and would shift an external memory location rather than a CPU register. Strictly speaking, the seven registers that may be specified result in seven unique instructions, which could be viewed as seven Implied Addressing instructions.

The AND R instruction is shown in Fig. 4-8. Here the instruction is a one-byte instruction (because it was an 8080 one-byte instruc-

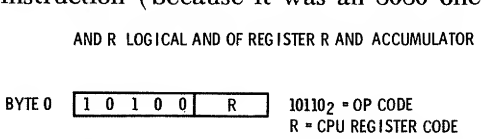


Fig. 4-8. Register addressing in AND R instruction.

tion) with the least significant three bits of the byte specifying the register to be used in the instruction. The coding of the registers is identical to the coding used in the RL R. AND R takes the contents of the specified R register (A, B, C, D, E, H, or L), logically ANDs it with the contents of the A register, and puts the result back into the A register. The condition codes are set on the result of the ANDing operation. As an example, the instruction shown in Fig. 4-9 would AND the contents of the D register with the A register contents and put the results in the A register.

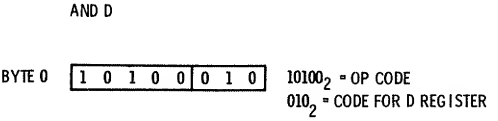


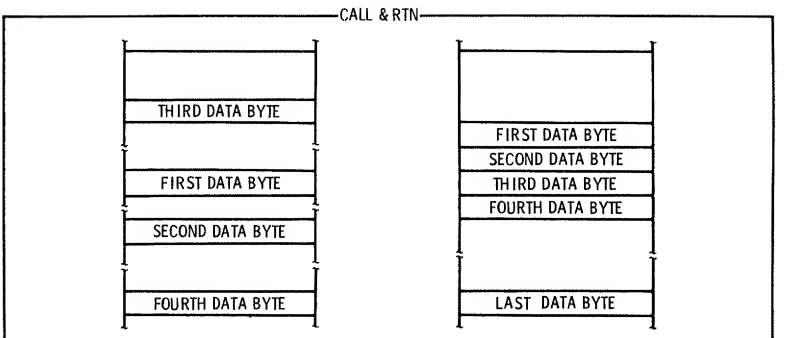
Fig. 4-9. Register addressing example.

Instruction groups that utilize this addressing mode would include the 8-Bit Arithmetic and Logical, 16-Bit Arithmetic, Rotate and Shift, and Bit Set, Reset, and Test groups.

REGISTER INDIRECT ADDRESSING

Instructions in this group include the original 8008 instructions that utilized the H and L register pair (*H*igh and *L*ow) as a memory address pointer. In the 8008, data in memory could only be addressed by the HL pointer. The 8080 added the capability to use register pairs B,C and D,E as pointers and also added the capability of Extended Addressing, where each memory location could be individually addressed. Register Indirect Addressing is a detriment where data must be addressed in random (noncontiguous) memory locations. When data is grouped in contiguous blocks, such as tables or strings, however, accessing data by the pointer method is somewhat more efficient. The reason for the inefficiency in accessing random memory locations is that the pointer register must be loaded with the address of the new byte of data to be accessed before each instruction of this kind is executed. Access of contiguous data is made simpler by instructions that automatically increment and decrement by one the register pairs used as pointers. The two procedures for accessing blocks of random and contiguous data are shown in Table 4-1, along with the relative times. Note that the examples are for illustrative purposes only to point out the deficiencies in register indirect addressing; the Z-80 has more efficient ways to access data and they will be described later in this chapter.

Table 4-1. Data Access Using Register Indirect Addressing Charts



RANDOM ACCESS	SEQUENTIAL (CONTIGUOUS) ACCESS
1. LOAD DATA POINTER WITH ADDRESS OF NEXT DATA BYTE (5 UNITS). 2. LOAD BYTE USING REGISTER INDIRECT ADDRESSING (3, 5). 3. PROCESS DATA BYTE (X). 4. DONE? IF NOT, GO TO 1 (7). 5. DONE. X + 15, 5 UNITS/BYTE	1. LOAD DATA POINTER WITH START OF DATA. 2. LOAD BYTE USING REGISTER INDIRECT ADDRESSING (3, 5). 3. PROCESS DATA BYTE (X). 4. BUMP REGISTER POINTER BY 1 (2, 5). 5. DONE? IF NOT, GO TO 2 (7). 6. DONE. X + 13 UNITS/BYTE

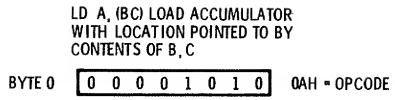


Fig. 4-10. Register indirect addressing in LD A,(BC) instruction.

Examples of the instruction format for this way of addressing are shown for an LD A,(BC) instruction (Fig. 4-10) and an INC (HL) instruction (Fig. 4-11). The LD A,(BC) is a one-byte instruction that loads the contents of the memory location pointed to by register pair BC into the A register. No condition codes are affected. The INC (HL) instruction increments the contents of the memory location pointed to by the HL register pair by one. The condition codes are set on the results of the increment.

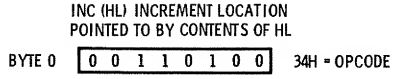


Fig. 4-11. Register indirect addressing in INC (HL) instruction.

When register indirect addressing is employed, the register pairs utilized as pointers hold the memory address as a 16-bit address as one would expect:

Register Pair	Most Significant Byte	Least Significant Byte
B,C	B	C
D,E	D	E
H,L	H(igh)	L(ow)
SP	SP bits 15-8	SP bits 7-0

Register indirect addressing is primarily used for 8008 compatible instruction groups such as the 8-Bit Load, 8-Bit Arithmetic and Logical, and Rotate-Shift groups.

EXTENDED ADDRESSING

The extended addressing instructions hold the address of the data in the instruction itself, in a fashion similar to many minicomputers and larger machines. Although this means that the instruction word is longer, all locations in memory can be addressed directly, and this mode is many times called direct addressing. The format of this kind of addressing is shown for an LD A,(NN) instruction and an LD (NN),HL instruction.

The LD A,(NN) is a classical computer instruction shown in Fig. 4-12. Bytes 1 and 2 of the instruction specify a location in memory. The 8-bit contents of this location are loaded into the accumulator. No condition codes are affected. Byte 1 of the address is the least significant byte, while byte 2 is most significant.

The LD (NN),HL instruction is an extended addressing instruction that does the opposite of the first example. It takes the contents of register pair H,L and stores it into the memory location specified in bytes 1 and 2 of the instruction (see Fig. 4-13). Just as in all instructions like this, the address of the memory location is ordered the least significant byte (byte 1) followed by the most significant

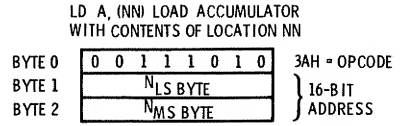


Fig. 4-12. Extended addressing in LD A,(NN) instruction.

byte (byte 2). The contents of the L register are stored in memory location NN and the contents of the H register are stored in memory location NN+1. An interesting thing to note about instructions like these that move data from CPU registers to memory is that Zilog chose to refer to them as LDs or *Loads*, when the usual mnemonic is ST for *Stores*. This classification may be rather confusing until one has worked with the mnemonics for some time.

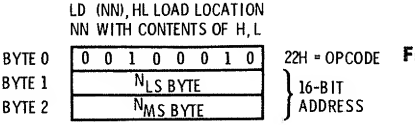


Fig. 4-13. Extended addressing in LD (NN),HL instruction.

Note that the 16-bit address in the instruction can address 2^{16} or 65,536 memory locations. The size of the address field in this instruction format together with the 16-bit width of the register pairs are the primary limitations to the size of external memory that can be employed without special memory *banking* schemes. Extended addressing is used primarily for instructions in the 8- and 16-bit Load groups.

MODIFIED PAGE ZERO ADDRESSING

This addressing mode is used only for one instruction, the RST P or *Restart Page Zero* instruction. The effect of this instruction is to cause a branch to one of eight *page 0* locations after pushing the current contents of the program counter into the stack. Page 0 in the Z-80 as in other computers is defined as the area of external memory that can be addressed in 8 bits. Since $2^8 = 256$, memory locations 0 through 255 constitute page zero. The format of the RST P is shown in Fig. 4-14. The T field in the instruction is three bits wide. Depending on the configuration of bits in the T field, a branch may be made to locations 0H, 8H, 10H, 18H, 20H, 28H,

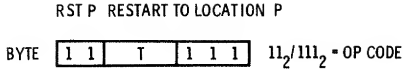


Fig. 4-14. Modified page zero addressing in restart instruction.

P	T
00H	0
08H	1
10H	2
18H	3
20H	4
28H	5
30H	6
38H	7

30H, or 38H as shown. This instruction is discussed more fully in the next chapter.

RELATIVE ADDRESSING

Relative addressing is primarily used in minicomputers or microcomputers to shorten instructions and reduce the amount of memory that programs occupy. If direct (extended) addressing is used to enable addressing all of memory, the address portion of the instruction is two bytes long (16 bits can address 64K). In both page zero and relative addressing, the address portion of the instruction is one byte long, reducing the instruction size from three bytes (op code plus address) to two bytes. Page zero addressing allows addressing only of page zero; relative addressing allows addressing of 256 memory locations grouped around the current instruction. Fig. 4-15 shows how this scheme is implemented. The second byte of the instruction is a signed value of -128_{10} to $+127_{10}$ (10000000_2 to 01111111_2). When this value is added to the current contents of the program counter, a memory location -126 to $+129$ bytes away is addressed since the program counter points to the instruction *after* the relative addressing instruction. As the current instruction moves through

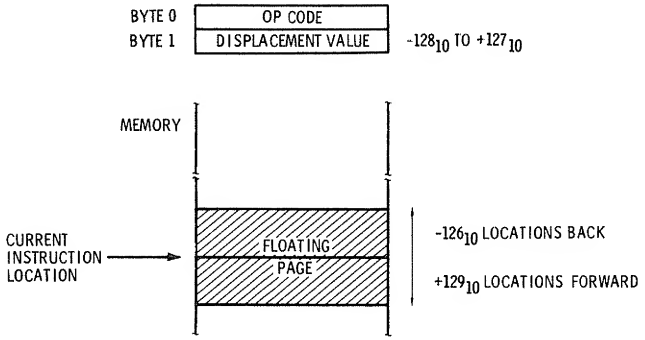


Fig. 4-15. Relative addressing.

memory, the block of memory that can be addressed moves (or floats) along with the current instruction. The premise for this manner of addressing is that in most cases it is sufficient to address memory in the immediate area of the current instruction; most programs *will* operate on data near the current instruction.

Relative addressing on the Z-80 is used only for the Jump Group of instructions, allowing conditional and unconditional jumps back up to 126 locations or forward 129 locations from the current instruction. An example of relative addressing for a jump is shown in Fig. 4-16.

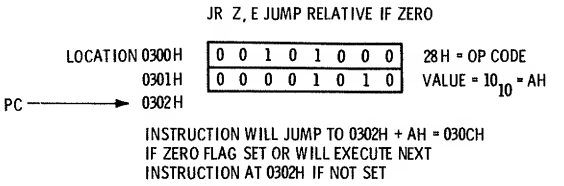


Fig. 4-16. Relative addressing in JR Z,E instruction.

INDEXED ADDRESSING

Indexed addressing is an addressing mode that permits using the two index registers in the Z-80, IX and IY. Many instruction groups permit using the indexed addressing mode and it is one of the most powerful features that the Z-80 offers. The format of this addressing mode is shown in Fig. 4-17. The op code of the instruction is in bytes 0 and 1; while the third byte holds an 8-bit *signed* displacement of -128_{10} through $+127_{10}$. This displacement is added to the contents of the specified index register IX or IY to determine the *effective address* of the memory operand.

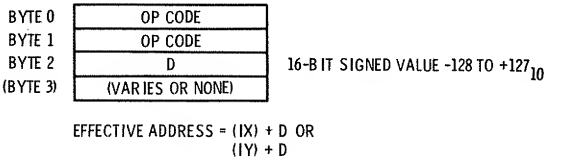


Fig. 4-17. Indexed addressing.

For example, consider the instruction LD (IY + D),N that uses the IY index register. This is shown in Fig. 4-18. The LD (IY + D),N loads (stores) the immediate value N into the memory location specified by the effective address. If the contents of IY are 1003H (the index registers are 16-bit registers), an LD (IY + D),N with a displacement field of 40H will store N into memory location 1043H.

The indexing operation is powerful because many programs must have the ability to process *tables* or *lists* of data in memory. Examples of the use of indexing are provided in section II. Instruction groups using the indexed addressing mode are the 8-Bit Load, 8-Bit

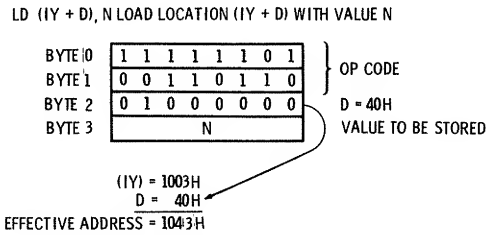


Fig. 4-18. Indexed addressing example.

Arithmetic and Logical, Rotate and Shift, and Bit, Set, Reset, and Test Groups.

BIT ADDRESSING

The last addressing group is the bit addressing group. Bit addressing is used in conjunction with the previous addressing modes to provide testing, setting, or resetting any one of the 8-bits in an operand. These operations would have to be performed by as many as three instructions in the 8080 or other computers. An example of this is provided for the SET B,(IX + D) instruction shown in Fig. 4-19. The SET instruction sets a specified bit, and in this case the address of the byte containing the bit to be set is given by (IX + D), an indexed addressing operation. The bit specified in the B field of the instruction will be set after the instruction has been executed. No condition codes are affected. The bit to be set is as follows:

B Field	Bit to be Set
000	XXXXXXX1
001	XXXXXX1X
010	XXXXX1XX
011	XXXX1XXX
100	XXX1XXXX
101	XX1XXXXX
110	X1XXXXXX
111	1XXXXXXX

Other examples of the bit addressing mode are shown in Fig. 4-20, which shows the "before" and "after" condition for various SET B,R instructions specifying a bit set for CPU register C.

Chart 4-1. Z-80 Addressing Modes

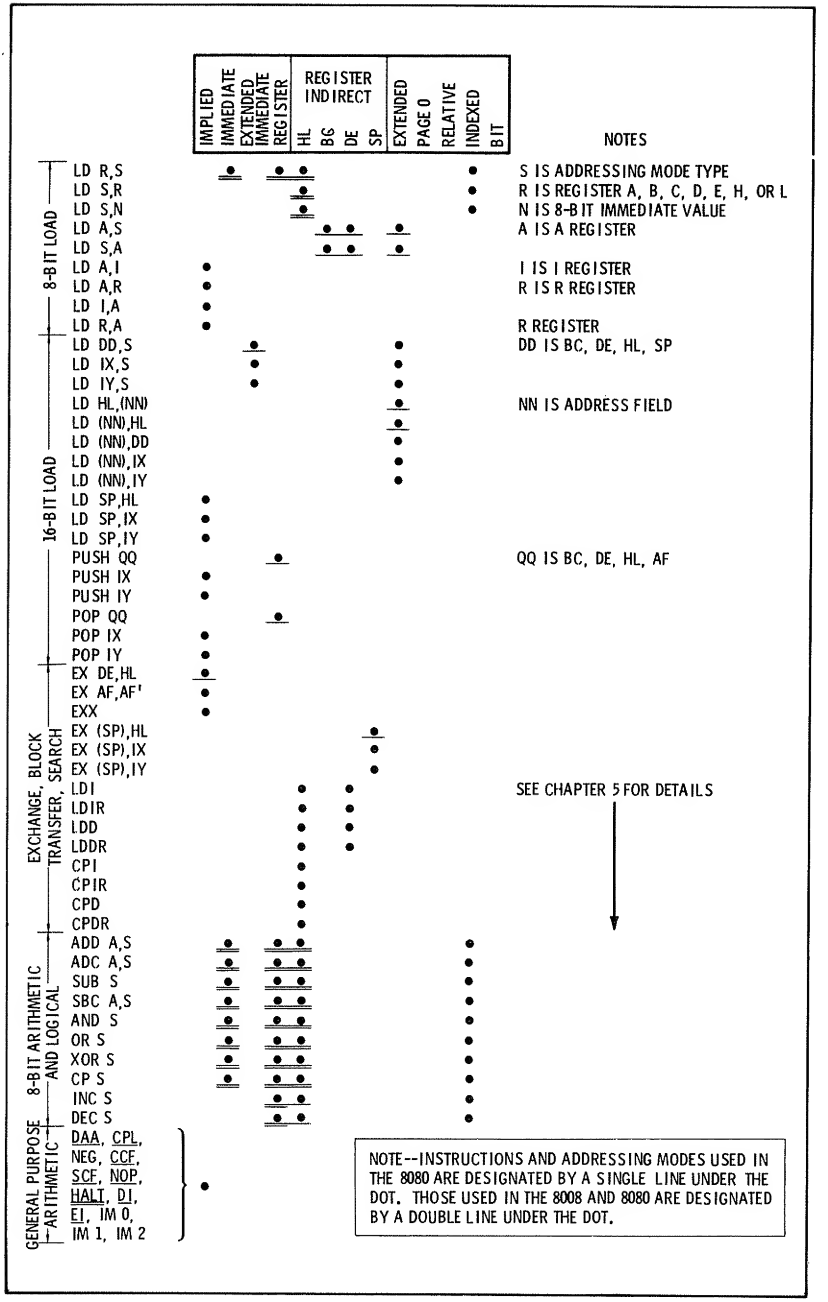
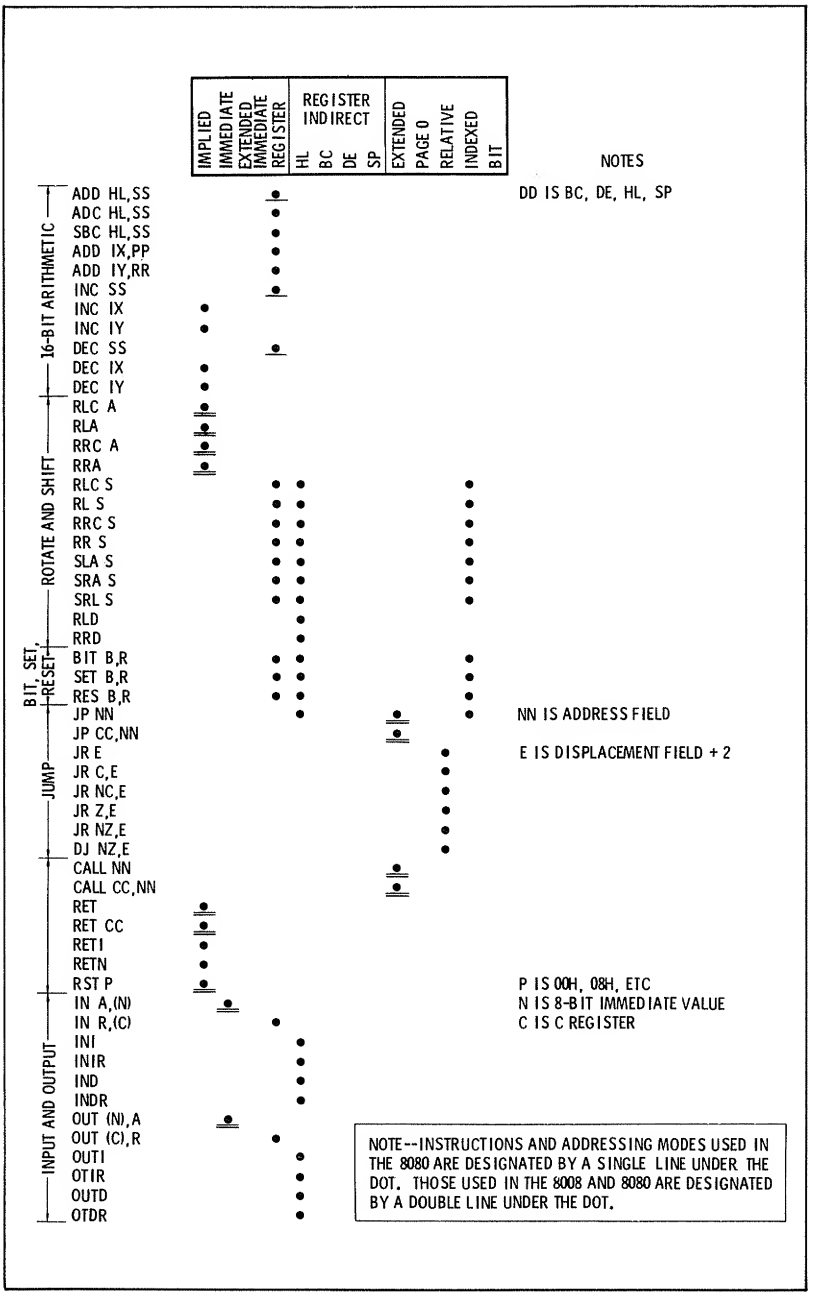


Chart 4-1. Z-80 Addressing Modes—cont



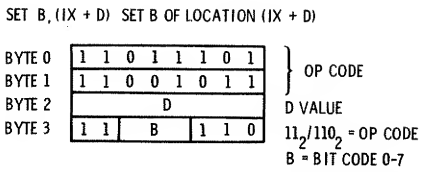


Fig. 4-19. Bit addressing/indexed addressing in SET B,(IX + D) instruction.

As the combinations of addressing modes employed in the various instructions can be almost overwhelming on first encounter, Chart 4-1 provides a reference chart for instruction groups. The chart follows the same notation as has been used in the above description

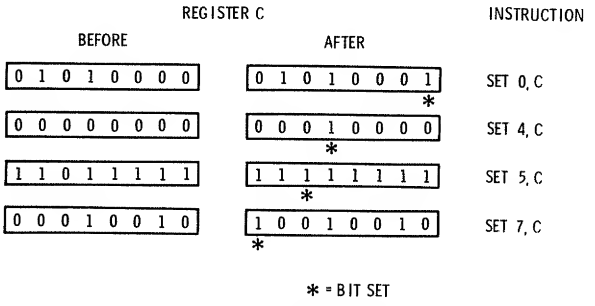


Fig. 4-20. Bit addressing example.

and that will be used in a discussion of the various instruction methods in the next chapter. Instructions and addressing modes used in the 8080 are designated by a single line under the dot. Those used in the 8008 and 8080 are designated by a double line under the dot.

CHAPTER 5

Instruction Set

The table of addressing modes given in Chapter 4 cross-references Z-80 instructions with its addressing modes. For discussion purposes, the instruction repertoire of the Z-80 may be classified into the groups shown in Chart 4-1. These groups are:

1. 8-Bit Load
2. 16-Bit Load
3. Exchange, Block Transfer, and Search
4. 8-Bit Arithmetic and Logical
5. General-Purpose Arithmetic and CPU Control
6. 16-Bit Arithmetic
7. Rotate and Shift
8. Bit Set, Reset, and Test
9. Jump
10. Call and Return
11. Input and Output

8-BIT LOAD GROUP

The 8-Bit Load Group is shown in Table 5-1. About half of the instructions in this group load an 8-bit value into a CPU register from another CPU register, immediate value in the instruction, or memory location. The other half of the instructions store an 8-bit value from a CPU register or immediate value into a CPU register or memory location. In all cases, the *source* register remains unchanged after the transfer.

Four of the instructions simply transfer the contents of the I and R registers into the current A register and vice versa. LD A,I loads

Table 5-1. 8-Bit Load Group

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T Cycles	Comments r, r′	Reg.
LD r,r′	r ← r′	•	•	•	•	•	•	01 r r′	1	1	4	r, r′	Reg.
LD r,n	r ← n	•	•	•	•	•	•	00 r 110	2	2	7	000	B
								← n →				001	C
LD r,(HL)	r ← (HL)	•	•	•	•	•	•	01 r 110	1	2	7	010	D
LD r,(IX+d)	r ← (IX + d)	•	•	•	•	•	•	11 011 101	3	5	19	011	E
								01 r 110				100	H
								← d →				101	L
LD r,(IY+d)	r ← (IY + d)	•	•	•	•	•	•	11 111 101	3	5	19	111	A
								01 r 110					
								← d →					
LD (HL),r	(HL) ← r	•	•	•	•	•	•	01 110 r	1	2	7		
LD (IX+d),r	(IX + d) ← r	•	•	•	•	•	•	11 011 101	3	5	19		
								01 110 r					
								← d →					
LD (IY+d),r	(IY + d) ← r	•	•	•	•	•	•	11 111 101	3	5	19		
								01 110 r					
								← d →					
LD (HL),n	(HL) ← n	•	•	•	•	•	•	00 110 110	2	3	10		
								← n →					
LD (IX+d),n	(IX + d) ← n	•	•	•	•	•	•	11 011 101	4	5	19		
								00 110 110					
								← d →					
								← n →					

LD (IY+d),n	(IY + d) ← n	●	●	●	●	●	●	11 111 101 00 110 110 ← d → ← n →	4	5		
LD A,(BC)	A ← (BC)	●	●	●	●	●	●	00 001 010	1	2	7	
LD A,(DE)	A ← (DE)	●	●	●	●	●	●	00 011 010	1	2	7	
LD A,(nn)	A ← (nn)	●	●	●	●	●	●	00 111 010 ← n → ← n →	3	4	13	
LD (BC),A	(BC) ← A	●	●	●	●	●	●	00 000 010	1	2	7	
LD (DE),A	(DE) ← A	●	●	●	●	●	●	00 010 010	1	2	7	
LD (nn),A	(nn) ← A	●	●	●	●	●	●	00 110 010 ← n → ← n →	3	4	13	
LD, A,I	A ← I	●	↕	IFF	↕	0	0	11 101 101 01 010 111	2	2	9	
LD A,R	A ← R	●	↕	IFF	↕	0	0	11 101 101 01 011 111	2	2	9	
LD, I,A	I ← A	●	●	●	●	●	●	11 101 101 01 000 111	2	2	9	
LD R,A	R ← A	●	●	●	●	●	●	11 101 101 01 001 111	2	2	9	

Notes:

r, r' means any of the registers A, B, C, D, E, H, L

IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag.

Flag Notation:

● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,

↕ = flag is affected according to the result of the operation.

the A register with the contents of the interrupt Vector Register I. LD A,R loads the A register with the contents of the Memory Refresh Register R. LD I,A and LD R,A do the reverse. No condition codes are affected for the latter two. The former two set the condition codes as shown. These four instructions do not exist in the 8080 or 8008 as neither microprocessor had the I or R registers.

The LD R,S instructions load the specified CPU register in the R field with the contents of another CPU register (LD R,R'), an 8-bit immediate value (LD R,N), or an 8-bit value from a memory location [LD R,(HL); LD R,(IX+D); LD R,(IY+D)]. None of the condition-code bits are affected after the load. LD S,R does the opposite of LD R,S, that is, the contents of a CPU register R is transferred to a memory location using either an HL register pointer method of addressing [LD (HL),R] or indexed addressing [LD (IX+D),R or LD (IY+D),R]. This is in fact a "store" kind of instruction (called a MOV in the 8080 and 8008). LD S,N is similar

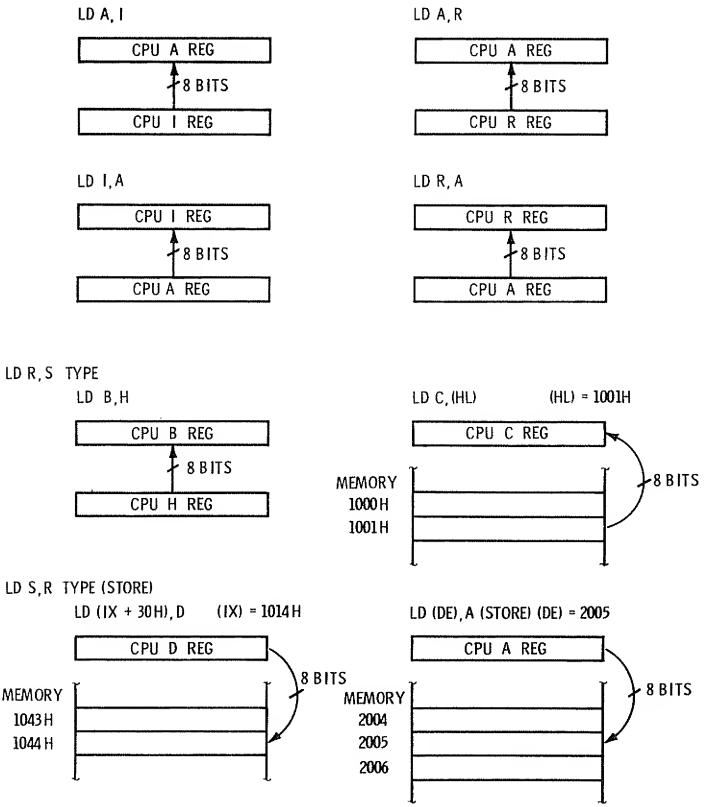


Fig. 5-1. Eight-bit load group examples.

except that an immediate value is stored into a memory location [LD (HL),N; LD(IX+D),N; or LD (IY+D),N]. None of the condition codes are affected by the load (store).

The last instructions of this group load or store the A register only with a memory location specified by register pointers BC, DE, or by an extended (direct) addressing. A is loaded by LD A,(BC); LD A,(DE); or LD A,(NN) and stored by LD (BC),A; LD (DE), A; and LD (NN),A. No condition codes are affected.

Examples of this group are shown in Fig. 5-1 which illustrates the various addressing modes and instruction types.

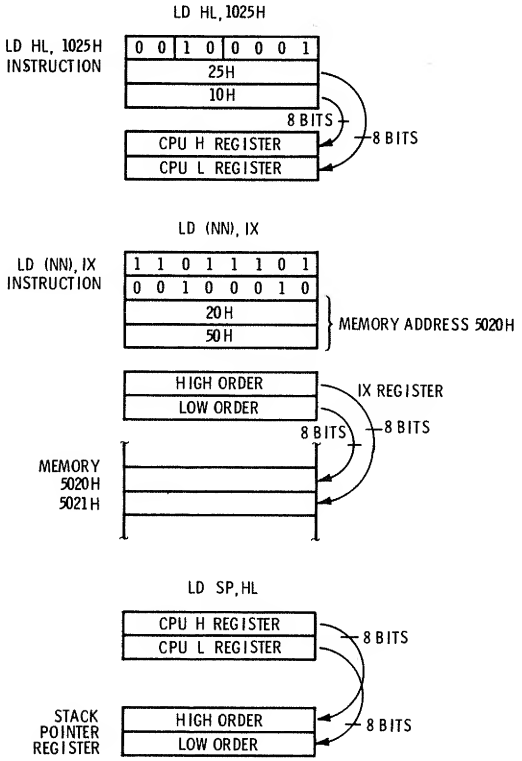


Fig. 5-2. Sixteen-bit load group examples.

16-BIT LOAD GROUP

This group allows any register pair BC, DE, HL, or SP, or the IX and IY registers to be loaded by an extended immediate instruction (LD DD,NN; LD IX,NN; or LD IY,NN). See Table 5-2. Here a 16-bit immediate value in the instruction is loaded into the selected

Table 5-2. 16-Bit Load Group

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
LD dd,nn	dd ← nn	●	●	●	●	●	●	00 dd0 001 ← n → ← n →	3	3	10	**dd Pair** 00 BC 01 DE 10 HL 11 SP
LD IX,nn	IX ← nn	●	●	●	●	●	●	11 011 101 00 100 001 ← n → ← n →	4	4	14	
LD IY,nn	IY ← nn	●	●	●	●	●	●	11 111 101 00 100 001 ← n → ← n →	4	4	14	
LD HL,(nn)	H ← (nn + 1) L ← (nn)	●	●	●	●	●	●	00 101 010 ← n → ← n →	3	5	16	
LD, dd,(nn)	dd_H ← (nn + 1) dd_L ← (nn)	●	●	●	●	●	●	11 101 101 01 dd1 011 ← n → ← n →	4	6	20	
LD IX,(nn)	IX_H ← (nn + 1) IX_L ← (nn)	●	●	●	●	●	●	11 011 101 00 101 010 ← n → ← n →	4	6	20	
LD IY,(nn)	IY_H ← (nn + 1) IY_L ← (nn)	●	●	●	●	●	●	11 111 101 00 101 010 ← n → ← n →	4	6	20	
LD (nn),HL	(nn + 1) ← H (nn) ← L	●	●	●	●	●	●	00 100 010 ← n → ← n →	3	5	16	

LD (nn),dd	(nn + 1) ← dd_H (nn) ← dd_L	●	●	●	●	●	●	11 101 101 01 dd0 011 ← n → ← n →	4	6	20		
LD (nn),IX	(nn + 1) ← IX_H (nn) ← IX_L	●	●	●	●	●	●	11 011 101 00 100 010 ← n → ← n →	4	6	20		
LD (nn),IY	(nn + 1) ← IY_H (nn) ← IY_L	●	●	●	●	●	●	11 111 101 00 100 010 ← n → ← n →	4	6	20		
LD SP,HL	SP ← HL	●	●	●	●	●	●	11 111 001	1	1	6		
LD SP,IX	SP ← IX	●	●	●	●	●	●	11 011 101 11 111 001	2	2	10		
LD SP,IY	SP ← IY	●	●	●	●	●	●	11 111 101 11 111 001	2	2	10		
PUSH qq	(SP — 2) ← qq_L (SP — 1) ← qq_H	●	●	●	●	●	●	11 qq0 101	1	3	11	**qq**	**Pair**
PUSH IX	(SP — 2) ← IX_L (SP — 1) ← IX_H	●	●	●	●	●	●	11 011 101 11 100 101	2	4	15	00 01 10 11	BC DE HL AF
PUSH IY	(SP — 2) ← IY_L (SP — 1) ← IY_H	●	●	●	●	●	●	11 111 101 11 100 101	2	4	15		
POP qq	qq_H ← (SP + 1) qq_L ← (SP)	●	●	●	●	●	●	11 qq0 001	1	3	10		
POP IX	IX_H ← (SP + 1) IX_L ← (SP)	●	●	●	●	●	●	11 011 101 11 100 001	2	4	14		
POP IY	IY_H ← (SP + 1) IY_L ← (SP)	●	●	●	●	●	●	11 111 101 11 100 001	2	4	14		

Notes:
dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair, respectively.
E.g., BC_L = C, AF_H = A.

Flag Notation:
● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,

Table 5-3. Exchange Group and Block Transfer and Search Group

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76	543	210	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE,HL	DE ↔ HL	●	●	●	●	●	●	11	101	011	1	1	4	
EX AF,AF'	AF ↔ AF'	●	●	●	●	●	●	00	001	000	1	1	4	
EXX	(BC ↔ BC', DE ↔ DE', HL ↔ HL')	●	●	●	●	●	●	11	011	001	1	1	4	Register bank and auxiliary register bank exchange
EX (SP),HL	H ↔ (SP + 1) L ↔ (SP)	●	●	●	●	●	●	11	100	011	1	5	19	
EX (SP),IX	IX_H ↔ (SP + 1) IX_L ↔ (SP)	●	●	●	●	●	●	11 11	011 100	101 011	2	6	23	
EX (SP),IY	IY_H ↔ (SP + 1) IY_L ↔ (SP)	●	●	●	●	●	●	11 11	111 100	101 011	2	6	23	
LDI	(DE) ← (HL) DE ← DE + 1 HL ← HL + 1 BC ← BC − 1	●	●	① ↕	●	0	0	11 10	101 100	101 000	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE) ← (HL) DE ← DE + 1 HL ← HL + 1 BC ← BC − 1 Repeat until BC = 0	●	●	0 ①	●	0	0	11 10	101 110	101 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE) ← (HL) DE ← DE − 1 HL ← HL − 1 BC ← BC − 1	●	●	① ↕	●	0	0	11 10	101 101	101 000	2	4	16	

LDDR	(DE) ← (HL) DE ← DE — 1 HL ← HL — 1 BC ← BC — 1 Repeat until BC = 0	•	•	0	•	0	0	11 101 101 10 111 000		2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A — (HL) HL ← HL + 1 BC ← BC — 1	•	② ↕	① ↕	↕	1	↕	11 101 101 10 100 001		2	4	16	
CPIR	A — (HL) HL ← HL + 1 BC ← BC — 1 Repeat until A = (HL) or BC = 0	•	② ↕	① ↕	↕	1	↕	11 101 101 10 110 001		2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A — (HL) HL ← HL — 1 BC ← BC — 1	•	② ↕	① ↕	↕	1	↕	11 101 101 10 101 001		2	4	16	
CPDR	A — (HL) HL ← HL — 1 BC ← BC — 1 Repeat until A = (HL) or BC = 0	•	② ↕	① ↕	↕	1	↕	11 101 101 10 111 001		2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes:

① P/V flag is 0 if the result of BC — 1 = 0, otherwise P/V = 1

② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation:

• = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,

↕ = flag is affected according to the result of the operation.

register pair, IX, or IY. Any register pair IX or IY can also be loaded or stored directly (extended addressing mode) by LD DD,(NN); LD IX,(NN); LD IY,(NN); LD (NN),HL; LD (NN),DD; LD (NN),IX; or LD (NN),IY.

The contents of HL, IX, or IY can be transferred to the SP register by LD SP,HL; LD SP,IX; or LD SP,IY.

The remaining instructions in this group allow 16-bit register pairs BC, DE, HL, or AF (A register and flags) to be pushed onto or pulled from the stack. Fig. 5-2 shows examples of the use of these instructions.

EXCHANGE, BLOCK TRANSFER, AND SEARCH GROUP

The exchange instructions in this group allow various exchanges of 16 bits of data between register pairs in the *same* set of registers and exchanges between the *two* sets of registers (see Table 5-3).

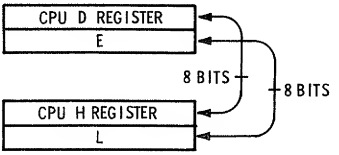


Fig. 5-3. EX DE, HL instructions.

EX DE,HL simply exchanges the contents of register pairs DE and HL in the current set of registers as shown in Fig. 5-3. EX AF,AF′, however, exchanges the contents of the A register and flag register of the *current* set of registers and the inactive set of registers as shown in Fig. 5-4. EX X swaps the contents of the current set of BC, DE, and HL with the inactive set of BC′, DE′, and HL′ as shown in the same figure. No condition codes are affected in any of the above instructions. These instructions permit switching back and forth between the two sets of CPU registers with one or two instructions.

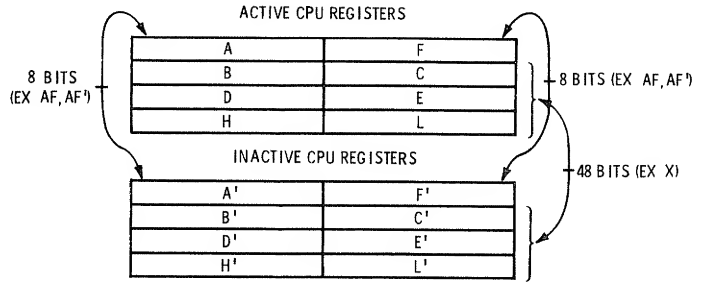


Fig. 5-4. EX AF,AF′; EX X instructions.

The three additional exchange instructions operate using the SP register as a pointer to the stack area. The stack pointer is not affected by execution of the instructions. Either HL, IX, or IY may be exchanged with current *top of stack* by instructions EX (SP),HL; EX (SP),IX; or EX (SP),IY. Examples of the three kinds of exchanges are shown in Fig. 5-5.

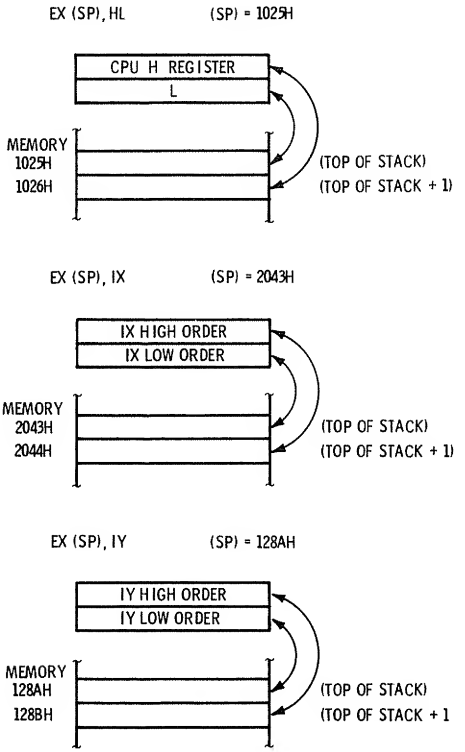


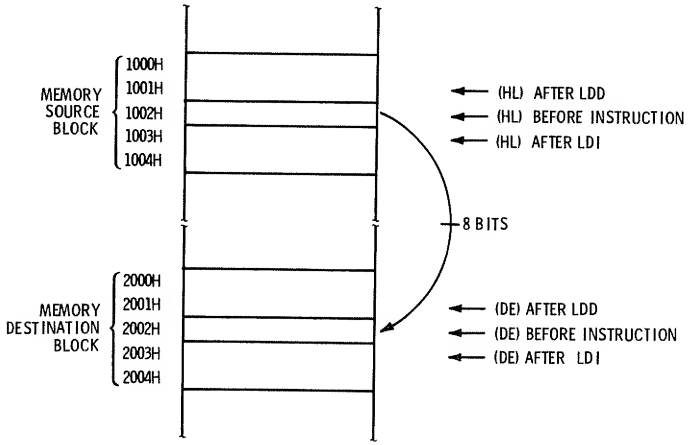
Fig. 5-5. EX (SP) instructions.

LDI, LDIR, LDD, and LDDR are four block transfer instructions that use register pairs BC, DE, and HL. All four instructions transfer a block of data from one place in memory to another. The block may be 1 to 64K bytes. Register pair BC must be preset with the number of bytes to be transferred, register pair HL must point to the starting address of the source block, and register pair DE must point to the starting address of the destination block. Instruction LDI performs the following actions when executed:

1. A byte is transferred from the source block to the destination block using registers HL and DE as pointers.

2. The HL and DE registers are incremented by one to point to the next byte of each block.
3. The *byte count* in BC is decremented by one.
4. If (BC) $\neq 0$, the P/V bit in the flags is set.

Instruction LDD performs the same functions as LDI except that the HL and DE registers in step 2 are *decremented* by one (LDI = *L*oad and *I*ncrement, while LDD = *L*oad and *D*ecrement). LDI, therefore, transfers data from block start to block end while LDD transfers data from block end to block start. The action of LDI and LDD are shown in Fig. 5-6.



LDI ACTIONS
1. TRANSFER BYTE FROM 1002H TO 2002H
2. ADD 1 TO HL TO POINT TO 1003H
3. ADD 1 TO DE TO POINT TO 2003H
4. SUBTRACT 1 FROM BC (BYTE COUNT)
5. GO ON TO NEXT INSTRUCTION

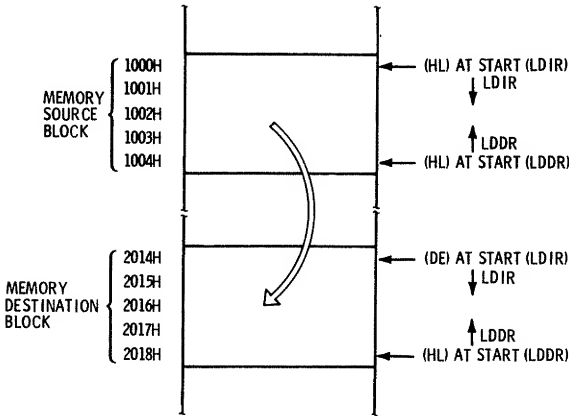
LDD ACTIONS
1. TRANSFER BYTE FROM 1002H TO 2002H
2. SUBTRACT 1 FROM HL TO POINT TO 1001H
3. SUBTRACT 1 FROM DE TO POINT TO 2001H
4. SUBTRACT 1 FROM BC (BYTE COUNT)
5. GO ON TO NEXT INSTRUCTION

Fig. 5-6. LDI and LDD instructions.

LDIR and LDDR perform identical functions to LDI and LDD with a supplemental action. If the byte count is not zero (P/V flag set), then the instruction continues transferring data until the byte count is 0. This means that there will be N executions of an LDIR or LDDR, where N is the initial value of the BC register. LDIR and LDDR are automatic transfers of a block of data while LDI and LDD are "semi-automatic," requiring a separate test of the P/V flag

for completion. Both are useful, as will be demonstrated in section II. Fig. 5-7 shows the actions of LDIR and LDDR.

The search instructions CPI, CPIR, and CPDR are similar to the block transfer instructions in that a block of memory locations is involved and these memory locations are scanned from start to end, or from end to start. The A register holds an 8-bit *search key* that can be 0 to 255. BC, as before, holds a byte count of 1 to 65535 and HL holds the starting address of the block (CPI or CPIR) or end-



LDIR ACTIONS

1. TRANSFER BYTE
2. ADD 1 TO HL
3. ADD 1 TO DE
4. SUBTRACT 1 FROM BC
5. IF (BC) ≠ 0 GO TO STEP 1
6. GO ON TO NEXT INSTRUCTION

} THESE ACTIONS REPEATED N TIMES WHERE N = # IN BC INITIALLY

LDDR ACTIONS

1. TRANSFER BYTE
2. SUBTRACT 1 FROM HL
3. SUBTRACT 1 FROM DE
4. SUBTRACT 1 FROM BC
5. IF (BC) ≠ 0 GO TO STEP 1
6. GO ON TO NEXT INSTRUCTION

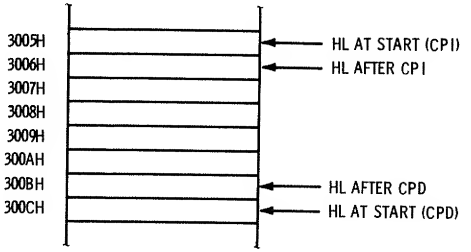
} THESE ACTIONS REPEATED N TIMES WHERE N = # IN BC INITIALLY

Fig. 5-7. LDIR and LDDR instructions.

ing address of the block (CPD or CPDR). When a CPI instruction is executed, the contents of the memory location addressed by HL is accessed and compared to the A register. If the memory byte equals the A register, flag Z is set in the condition codes. The byte count in BC is then decremented and the pointer in HL is incremented to point to the next memory location. CPD functions in the same manner except that the pointer in HL is decremented. CPI and CPD will search a block for a given byte semi-automatically as a test of the Z flag must be made after every execution of CPI

or CPD to determine whether the byte was found. Fig. 5-8 shows the actions of CPI and CPD.

CPIR and CPDR are similar to CPI and CPD except that they are fully automatic. If the byte count in BC is not equal to zero at the end of execution of the instruction, and the current memory byte does not equal the key value, the instruction is again executed for another comparison. The instruction is continually executed until either the byte count in BC is zero or until a memory location matches the key, as shown in Fig. 5-9.



CPI ACTIONS

1. READ NEXT BYTE
2. ADD 1 TO HL
3. SUBTRACT 1 FROM BC
4. COMPARE BYTE TO (A) AND SET FLAGS
5. GO ON TO NEXT INSTRUCTION

CPD ACTIONS

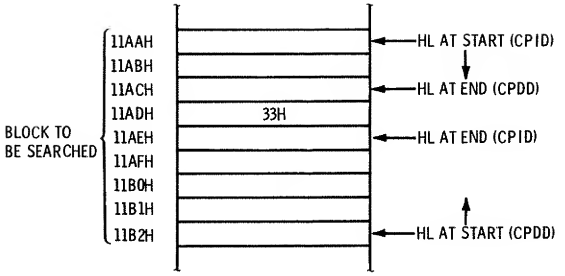
1. READ NEXT BYTE
2. SUBTRACT 1 FROM HL
3. SUBTRACT 1 FROM BC
4. COMPARE BYTE TO (A) AND SET FLAGS
5. GO ON TO NEXT INSTRUCTION

Fig. 5-8. CPI and CPD instructions.

8-BIT ARITHMETIC AND LOGICAL GROUP

The 8-bit arithmetic and logical instructions are used to add, subtract, AND, OR, exclusive OR, or compare two 8-bit operands, one of which must be in the A register. The second operand may be an immediate operand, may be in another CPU register, or may be in memory and referenced by HL register indirect addressing or by indexed addressing. The two operands are obtained, the designated function is performed, and the result goes into the A register. The condition codes are set as presented in Table 5-4.

There are two kinds of adds, ADD A,S and ADC A,S. In the first, the contents of the A register and the second operand are simply added and the results put into A; in the second, the contents of the A register, the second operand, and the current state of the carry flag are added and the results are put into the A register. The second



AT START

(HL) = 11AAH FOR CPIR 11B2H FOR CPDR
(BC) = 9
(A) = 33H

CPID ACTIONS

1. READ NEXT BYTE
2. ADD 1 TO HL
3. SUBTRACT 1 FROM BC
4. COMPARE BYTE TO (A) AND SET FLAGS
5. IF BC ≠ 0 AND BYTE ≠ A TO STEP 1
6. GO ON TO NEXT INSTRUCTION

} THESE ACTIONS REPEATED FOUR TIMES

CPDD ACTIONS

1. READ NEXT BYTE
2. SUBTRACT 1 FROM HL
3. SUBTRACT 1 FROM BC
4. COMPARE BYTE TO (A) AND SET FLAGS
5. IF BC ≠ 0 AND BYTE ≠ A GO TO STEP 1
6. GO ON TO NEXT INSTRUCTION

} THESE ACTIONS REPEATED SIX TIMES

Fig. 5-9. CPIR and CPDR instructions.

add permits multiple-precision addition and is discussed in Section II. Subtracts are analogous to the adds. SUB S subtracts the second operand from the contents of the A register, while SBC A,S subtracts the second operand and the current state of the carry from the contents of the A register. The add and subtract instructions are shown in three addressing mode examples in Fig. 5-10.

There are two additional instructions in this group, the INC S and DEC S instructions. They increment or decrement the contents of a CPU register (A, B, C, D, E, H, L) or memory location by one and set certain condition codes as listed in Table 5-4. As an immediate instruction makes no sense for this one-operand instruction only register, register indirect HL, and indexed addressing modes are permitted as shown in Fig. 5-11.

GENERAL-PURPOSE ARITHMETIC AND CPU CONTROL GROUP

The instructions in this group are listed in Table 5-5. They are all implied addressing instructions involving one or no operands. Two of the instructions involve one operand, CPL and NEG. Both

Table 5-4. 8-Bit Arithmetic and Logical Group

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76	543	210	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD r	A ← A + r	↕	↕	V	↕	0	↕	10	[000]	r	1	1	4	r Reg.
ADD n	A ← A + n	↕	↕	V	↕	0	↕	11	[000]	110	2	2	7	000 B
								←	n	→				001 C
ADD (HL)	A ← A + (HL)	↕	↕	V	↕	0	↕	10	[000]	110	1	2	7	010 D
ADD (IX+d)	A ← A + (IX + d)	↕	↕	V	↕	0	↕	11	011	101	3	5	19	011 E
								10	[000]	110				100 H
								←	d	→				101 L
ADD (IY+d)	A ← A + (IY + d)	↕	↕	V	↕	0	↕	11	111	101	3	5	19	111 A
								10	[000]	110				
								←	d	→				
ADC s	A ← A + s + CY	↕	↕	V	↕	0	↕		[001]					s is any of r, n, (HL), (IX + d) (IY + d) as shown for ADD instruction. The indicated bits replace the 000 in the ADD set above.
SUB s	A ← A − s	↕	↕	V	↕	1	↕		[010]					
SBC s	A ← A − s − CY	↕	↕	V	↕	1	↕		[011]					
AND s	A ← A ∧ s	0	↕	P	↕	0	↕		[100]					
OR s	A ← A ∨ s	0	↕	P	↕	0	↕		[110]					
XOR s	A ← A ⊕ s	0	↕	P	↕	0	↕		[101]					
CP s	A − s	↕	↕	V	↕	1	↕		[111]					
INC r	r ← r + 1	●	↕	V	↕	0	↕	00	r	[100]	1	1	4	
INC (HL)	(HL) ← (HL) + 1	●	↕	V	↕	0	↕	00	110	[100]	1	3	11	
INC (IX+d)	(IX + d) ← (IX + d) + 1	●	↕	V	↕	0	↕	11	011	101	3	6	23	
								00	110	[100]				
								←	d	→				

INC (IY+d)	(IY + d) ← (IY + d) + 1	●	↕	V	↕	0	↕	11 00 ←	111 110 d	101 100 →	3	6	23	
DEC d	d ← d — 1	●	↕	V	↕	1	↕			101				d is any of r, (HL), (IX + d), (IY + d) as shown for INC. Same format and states as INC. Replace 100 with 101 m OP code.

Notes:

The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow, P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation:

● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

CPL and NEG operate on the contents of the A register. CPL *ones-complements* the contents of the A register, changing all zeros to ones and all ones to zeros, as shown in Fig. 5-12. NEG *negates* the contents of the A register changing all zeros to ones and all ones to zeros and adding one as shown in the figure. The effect of CPL is to find the value $-[(A)+1]$ and NEG to find the value $-A$, where (A) is the previous contents of the A register. Condition codes are set as shown in Table 5-4.

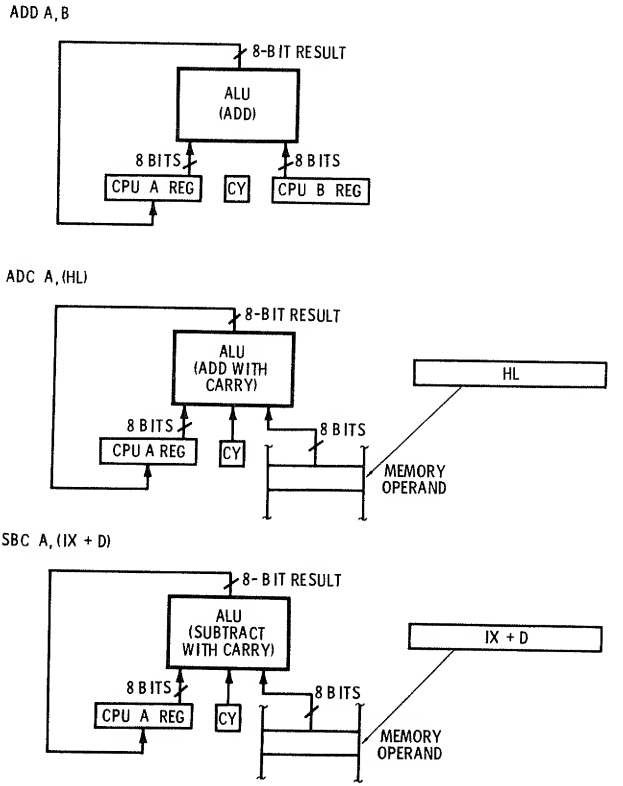


Fig. 5-10. Add and Subtract instruction examples.

Two of the instructions in this group operate on the carry (CY) flag of the condition codes. SCF sets the carry flag to a 1; CCF complements the current state of the carry – a 1 is set to a 0, and a 0 is set to a 1. These instructions are useful in setting the carry prior to arithmetic or shifting operations.

The NOP instruction does nothing and is used to "pad" a program area or is implemented automatically by the Z-80 during a HALT state to guarantee dynamic-memory refresh.

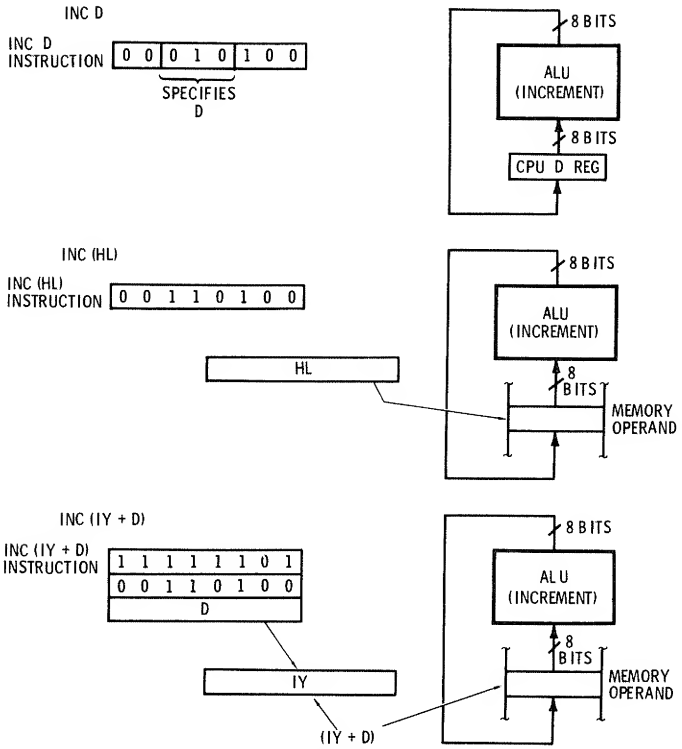


Fig. 5-11. INC and DEC instruction examples.

DI and EI disable or enable external (non-NMI) interrupts by resetting or setting the interrupt enable flip-flops IFF1 and IFF2. IM 0, IM 1, and IM 2 set interrupt modes 0, 1, or 2. The meaning of the various modes is discussed in Chapter 7.

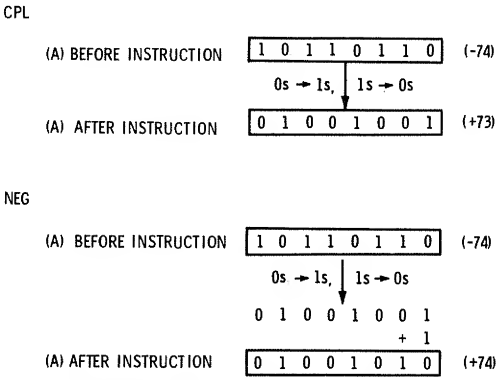


Fig. 5-12. CPL and NEG instruction examples.

Table 5-5. General-Purpose Arithmetic and CPU Control Groups

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
DAA	Converts acc. content into packed bcd following add or subtract with packed bcd operands	↕	↕	P	↕	●	↕	00 100 111	1	1	4	Decimal adjust accumulator
CPL	$A \leftarrow \overline{A}$	●	●	●	●	1	1	00 101 111	1	1	4	Complement accumulator (one's complement)
NEG	$A \leftarrow 0 - A$	↕	↕	V	↕	1	↕	11 101 101 01 000 100	2	2	8	Negate acc. (two's complement)
CCF	$CY \leftarrow \overline{CY}$	↕	●	●	●	0	X	00 111 111	1	1	4	Complement carry flag
SCF	$CY \leftarrow 1$	1	●	●	●	0	0	00 110 111	1	1	4	Set carry flag
NOP	No operation	●	●	●	●	●	●	00 000 000	1	1	4	
HALT	CPU halted	●	●	●	●	●	●	01 110 110	1	1	4	
DI	$IFF \leftarrow 0$	●	●	●	●	●	●	11 110 011	1	1	4	
EI	$IFF \leftarrow 1$	●	●	●	●	●	●	11 111 011	1	1	4	
IM 0	Set interrupt mode 0	●	●	●	●	●	●	11 101 101 01 000 110	2	2	8	
IM 1	Set interrupt mode 1	●	●	●	●	●	●	11 101 101 01 010 110	2	2	8	
IM 2	Set interrupt mode 2	●	●	●	●	●	●	11 101 101 01 011 110	2	2	8	

Notes:
IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

Flag Notation:
● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

The last instruction in this group is the DAA instruction. DAA, or Decimal Adjust Accumulator, allows the Z-80 to perform binary-coded decimal (bcd) addition or subtraction. (The 8080 can perform only bcd addition automatically.) The DAA is performed directly after an ADD, ADC, INC, SUB, SBC, DEC, or NEG and changes the binary results of the operation into bcd results. Bcd addition will be discussed in detail in Section II.

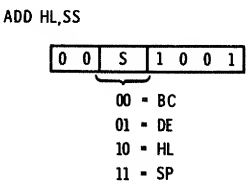


Fig. 5-13. Sixteen-bit arithmetic register encoding.

16-BIT ARITHMETIC GROUP

All of the instructions in this group operate on 16-bit double-precision values in either register pairs BC, DE, or HL, or in 16-bit

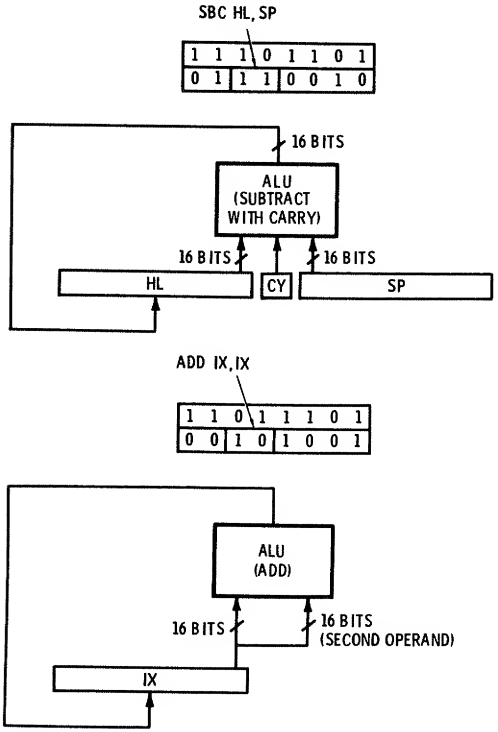


Fig. 5-14. Sixteen-bit arithmetic instruction examples.

Table 5-6. 16-Bit Arithmetic Group

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments ss	Reg.
ADD HL,ss	HL ← HL + ss	↕	●	●	●	0	X	00 ss1 001	1	3	11		00 BC
ADC HL,ss	HL ← HL + ss + CY	↕	↕	V	↕	0	X	11 101 101 01 ss1 010	2	4	15		01 DE
SBC HL,ss	HL ← HL − ss − CY	↕	↕	V	↕	1	X	11 101 101 01 ss0 010	2	4	15		10 HL
ADD IX,pp	IX ← IX + pp	↕	●	●	●	0	X	11 011 101 00 pp1 001	2	4	15	**pp**	**Reg.**
ADD IY,rr	IY ← IY + rr	↕	●	●	●	0	X	11 111 101 00 rr1 001	2	4	15	**rr**	**Reg.**
INC ss	ss ← ss + 1	●	●	●	●	●	●	00 ss0 011	1	1	6		
INC IX	IX ← IX + 1	●	●	●	●	●	●	11 011 101 00 100 011	2	2	10		
INC IY	IY ← IY + 1	●	●	●	●	●	●	11 111 101 00 100 011	2	2	10		
DEC ss	ss ← ss − 1	●	●	●	●	●	●	00 ss1 011	1	1	6		
DEC IX	IX ← IX − 1	●	●	●	●	●	●	11 011 101 00 101 011	2	2	10		
DEC IY	IY ← IY − 1	●	●	●	●	●	●	11 111 101 00 101 011	2	2	10		

Reg.
00 BC
01 DE
10 HL
11 SP

Reg.
00 BC
01 DE
10 IX
11 SP

Reg.
00 BC
01 DE
10 IY
11 SP

Notes:
ss is any of the register pairs BC, DE, HL, SP
pp is any of the register pairs BC, DE, IX, SP
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation:
● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

CPU registers IX, IY, or SP. Increments and decrements of BC, DE, HL, SP, IX, or IY can be performed by INC SS, INC IX, INC IY, DEC SS, DEC IX, or DEC IY. The SS-type instructions increment or decrement BC, DE, HL, or SP depending on the SS field of the instruction as shown in Fig. 5-13. The remaining increment and decrements are all implied addressing types.

Three of the instructions in this group permit adding, adding with carry, or subtracting with carry. The contents of BC, DE, HL, or SP can operate on the contents of the HL register with the result going to the HL register. The condition codes are set as shown in Table 5-6, and an example of the instructions is shown in Fig. 5-14. ADD IX,PP and ADD IY,RR permit addition of BC, DE, SP, or the same index register to IX and IY, respectively. The condition codes are set as listed in the table, and an example of the instruction is shown in the figure.

ROTATE AND SHIFT GROUP

The instructions in this group include the 8080 (8008) instructions that rotated only the A register and new instructions to shift A, B, C, D, E, H, or L or a memory operand in just about every possible shift configuration. Table 5-7 shows the rotate and shift instructions.

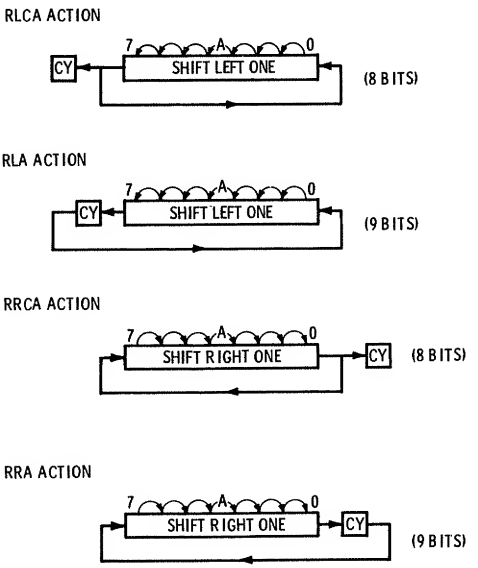
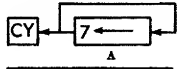
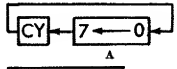
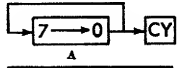
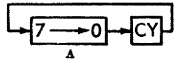
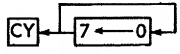
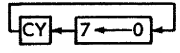
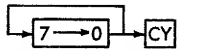
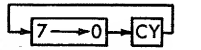
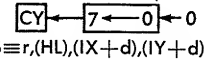
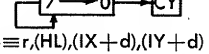
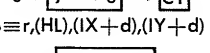
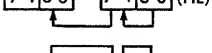
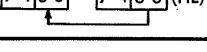


Fig. 5-15. RLCA, RLA, RRCA, RRA instructions.

Table 5-7. Rotate and Shift Group

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
RLCA		↕	●	●	●	0	0	00 000 111	1	1	4	Rotate left circular accumulator
RLA		↕	●	●	●	0	0	00 010 111	1	1	4	Rotate left accumulator
RRCA		↕	●	●	●	0	0	00 001 111	1	1	4	Rotate right circular accumulator
RRA		↕	●	●	●	0	0	00 011 111	1	1	4	Rotate right accumulator
RLC r		↕	↕	P	↕	0	0	11 001 011 00 000 r	2	2	8	Rotate left circular register r
RLC (HL)		↕	↕	P	↕	0	0	11 001 011 00 000 110	2	4	15	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
RLC (IX+d)	r,(HL),(IX+d),(IY+d)							11 011 101 11 001 011 ← d → 00 000 110	4	6	23	
RLC (IY+d)		↕	↕	P	↕	0	0	11 111 101 11 001 011 ← d → 00 000 110	4	6	23	
RL s		↕	↕	P	↕	0	0	010				Instruction format and states are as shown for RLC,s. To form new
	S≡r,(HL),(IX+d),(IY+d)											

RRC s	 S≡r,(HL),(IX+d),(IY+d)	↕	↕	P	↕	0	0	001					OP-code replace 000 of RLC,s with shown code
RR s	 S≡r,(HL),(IX+d),(IY+d)	↕	↕	P	↕	0	0	011					
SLA s	 S≡r,(HL),(IX+d),(IY+d)	↕	↕	P	↕	0	0	100					
SRA s	 S≡r,(HL),(IX+d),(IY+d)	↕	↕	P	↕	0	0	101					
SRL s	 S≡r,(HL),(IX+d),(IY+d)	↕	↕	P	↕	0	0	111					
RLD		●	↕	P	↕	0	0	11 01	101 101	101 111	2	5	18	Rotate digit left and right between the accumulator and location (HL) The content of the upper half of the accumulator is unaffected
RRD		●	↕	P	↕	0	0	11 01	101 100	101 111	2	5	18	

Flag Notation:
● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

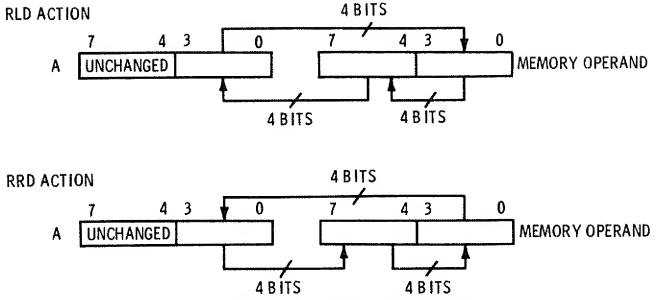


Fig. 5-16. RLD and RRD instructions.

RLCA, RLA, RRCA, and RRA rotate the A register only. The first letter of the mnemonic stands for *R*otate, the last *A*ccumulator, and the second the direction of the rotate, *l*eft or *r*ight. RLCA rotates left with the most significant bit going into the carry (CY) and the least significant bit position. RRCA performs a similar operation with a right shift. RLA and RRA perform a nine-bit shift with the previous contents of the carry shifting into the A register and the bit shifted out from the A register going into the carry. All four shifts are shown in Fig. 5-15.

Two shifts of this group RLD and RRD operate on the contents of a memory location, addressed by register indirect addressing HL, and the A register, and shifts four bits at a time. These two shifts are implemented to facilitate bcd operations, where each bcd digit is made up of four bits. If the reader considers bits 7-4 of the A register or memory location bcd digit position 0 and bits 3-0 bcd digit position 1, then these shifts are somewhat easier to follow. RLD shifts the memory operand$_{BCD1}$ into memory operand $_{BCD0}$ and memory operand$_{BCD0}$ into A_{BCD1}. The previous contents of memory operand$_{BCD1}$ are replaced by A_{BCD1} as shown in Fig. 5-16. Instruc-

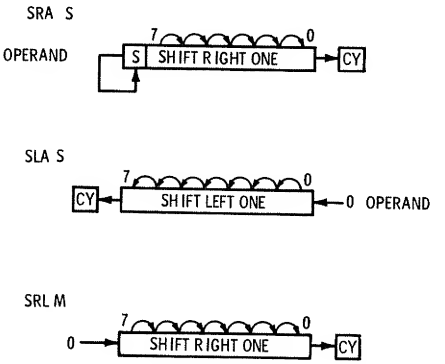


Fig. 5-17. SRA, SLA, SRL instructions.

tion RRD operates in the reverse direction as shown in the illustration. The condition codes are set as shown in Table 5-6.

The remaining shifts in this group operate either on CPU registers or on a memory location addressed by register indirect HL addressing or indexed addressing. Those with a mnemonic starting with an R are rotates, and those with a mnemonic starting with an S are arithmetic (SLA S, SRA S) or logical (SRL S). SLA S and SRA S perform arithmetic left and right shifts. Arithmetic shifts *sign-extend* the sign bit to the right on a right shift and sometimes retain the sign bit on a left shift. The Z-80 SRA S does extend the sign bit on a right shift as shown in Fig. 5-17, but does not retain it on a left shift.

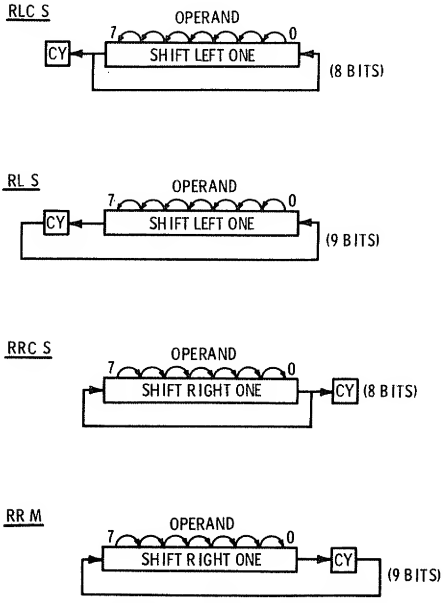


Fig. 5-18. RLC, RL, RRC, RR instructions.

Any of the seven current CPU registers can be shifted when register addressing is used with the R field specifying the register as shown in Fig. 5-17. The condition codes are set as listed in Table 5-6. Instruction SRL S performs a logical right shift with a zero going into the sign bit position. Note that for all three shifts a zero is shifted *into* the operand and that the carry is set by the bit shifting *out of* the operand.

Shifts RLC, RL, RRC, and RR are rotate shifts performing either an 8-bit shift (operand without carry), or a nine-bit shift (operand with carry). RLC and RRC rotate in 8-bit fashion, while RL and RR rotate in 9-bit fashion. All four shifts are shown in Fig. 5-18.

Table 5-8. Bit Set, Reset, and Test Group

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76	543	210	No. of Bytes	No. of M Cycles	No. of T States
BIT b,r	$Z \leftarrow \overline{r_b}$	●	↕	X	X	0	1	11	001	011	2	2	8
								01	b	r			
BIT b,(HL)	$Z \leftarrow \overline{(HL)_b}$	●	↕	X	X	0	1	11	001	011	2	3	12
								01	b	110			
BIT b,(IX+d)	$\leftarrow \overline{(IX+d)_b}$	●	↕	X	X	0	1	11	011	101	4	5	20
								11	001	011			
								←	d	→			
								01	b	110			
BIT b,(IY+d)	$Z \leftarrow (IY+d)_b$	●	↕	X	X	0	1	11	111	101	4	5	20
								11	001	011			
								←	d	→			
								01	b	110			
SET b,r	$r_b \leftarrow 1$	●	●	●	●	●	●	11	001	011	2	2	8
								[11]	b	r			
SET b,(HL)	$(HL)_b \leftarrow 1$	●	●	●	●	●	●	11	001	011	2	4	15
								[11]	b	110			
SET b,(IX+d)	$(IX+d)_b \leftarrow 1$	●	●	●	●	●	●	11	011	101	4	6	23
								11	001	011			
								←	d	→			
								[11]	b	110			

Comments	
r	Reg.
000	B
001	C
010	D
011	E
100	H
101	L
111	A
b	**Bit Tested**
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

SET b,(IY+d)	$(IY + d)_b \leftarrow 1$	●	●	●	●	●	●	11 11 ← 11	111 001 d b	101 011 → 110	4	6	23	
RES b,s	$s_b \leftarrow 0$ $s \equiv r$,(HL), (IX + d), (IY + d)							10						To form new OP-code replace 11 of SET b,s with 10 . Flags and time states for SET instruction

Notes:
The notation s_b indicates bit b (0 to 7) or location s.

Flag Notation:
● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

BIT, SET, RESET, AND TEST GROUP

The instructions in this group set, reset, or test one of the eight bits in a CPU register (A, B, C, D, E, H, or L) or memory operand. Register, register indirect, or indexed addressing may be used (see Table 5-8). In all three types, the B field specifies which bit of the byte is to be operated on as follows:

BIT	B
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

BIT B,R tests the bit and sets the Z flag if the bit *is a zero* and resets the Z flag if the bit *is a 1*. SET sets the indicated bit and does

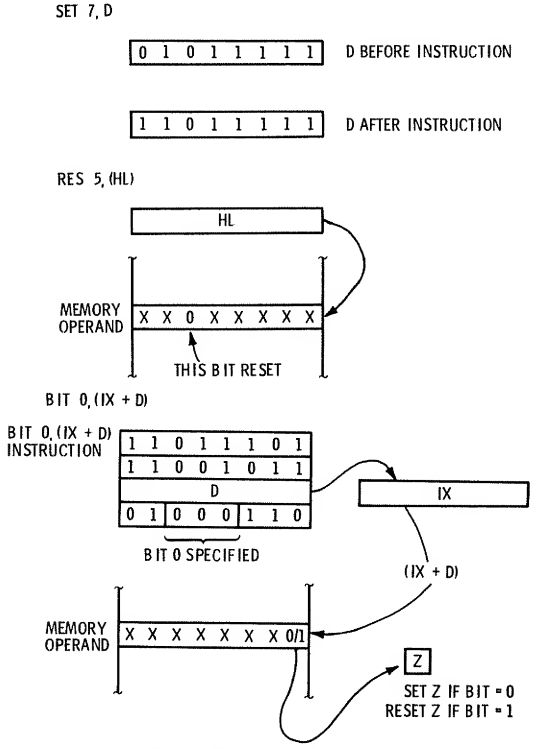


Fig. 5-19. SET, RES, BIT instruction examples.

Table 5-9. Jump, Call, and Return Group

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76	543	210	No. of Bytes	No. of M Cycles	No. of T States	Comments
JP nn	PC ← nn	●	●	●	●	●	●	11	000	011	3	3	10	
								←	n	→				
								←	n	→				
JP cc,nn	If condition cc is true PC ← nn, otherwise continue	●	●	●	●	●	●	11	cc	010	3	3	10	
								←	n	→				
								←	n	→				
JR e	PC ← PC + e	●	●	●	●	●	●	00	011	000	2	3	12	
								←	e − 2	→				
JR C,e	If C = 0, continue	●	●	●	●	●	●	00	111	000	2	2	7	If condition not met
								←	e − 2	→				
	If C = 1, PC ← PC + e										2	3	12	If condition is met
JR NC,e	If C = 1, continue	●	●	●	●	●	●	00	110	000	2	2	7	If condition not met
								←	e − 2	→				
	If C = 0, PC ← PC + e										2	3	12	If condition is met
JR Z,e	If Z = 0 continue	●	●	●	●	●	●	00	101	000	2	2	7	If condition not met
								←	e − 2	→				
	If Z = 1, PC ← PC + e										2	3	12	If condition is met
JR NZ,e	If Z = 1, continue	●	●	●	●	●	●	00	100	000	2	2	7	If condition not met
								←	e − 2	→				

cc	Condition	
000	NZ	nonzero
001	Z	zero
010	NC	noncarry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

Table 5-9. Jump, Call, and Return Group—cont

Mnemonic	Symbolic Operation	C	Z	P/V	S	N	H	76	543	210	No. of Bytes	No. of M Cycles	No. of T States	Comments
	If Z = 0, PC ← PC + e										2	3	12	If condition met
JP (HL)	PC ← HL	●	●	●	●	●	●	11	101	001	1	1	4	
JP (IX)	PC ← IX	●	●	●	●	●	●	11	011	101	2	2	8	
								11	101	001				
JP (IY)	PC ← IY	●	●	●	●	●	●	11	111	101	2	2	8	
								11	101	001				
DJNZ, e	B ← B—1	●	●	●	●	●	●	00	010	000	2	2	8	If B = 0
	If B = 0, continue							←	e — 2	→				
	If B ≠ 0, PC ← PC + e										2	3	13	If B ≠ 0
CALL c,nn	(SP—1) ← PC_H	●	●	●	●	●	●	11	001	101	3	5	17	
	(SP—2) ← PC_L							←	n	→				
	PC ← nn							←	n	→				
CALL cc, nn	If condition cc is false continue,	●	●	●	●	●	●	11	cc	100	3	3	10	If cc is false
								←	n	→				
	otherwise same as CALL nn							←	n	→	3	5	17	If cc is true
RET	PC_L ← (SP)	●	●	●	●	●	●	11	001	001	1	3	10	
	PC_H ← (SP + 1)													
RET cc	If condition cc is false continue,	●	●	●	●	●	●	11	cc	000	1	1	5	If cc is false
	otherwise same as RET										1	3	11	If cc is true

cc	Condition
000	NZ nonzero
001	Z zero

RETI	Return from interrupt	●	●	●	●	●	●	11 01	101 001	101 101	2	4	14
RETN	Return from nonmaskable interrupt	●	●	●	●	●	●	11 01	101 000	101 101	2	4	14
RST p	(SP—1) ← PC_H (SP — 2) ← PC_L PC_H ← 0 PC_L ← P	●	●	●	●	●	●	11	t	111	1	3	11

010	NC	noncarry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

t	P
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

Notes:

e represents the extension in the relative addressing mode.
e is a signed two's complement number in the range <—126, 129>
e — 2 in the op-code provides an effective address of pe + e as PC is incremented by 2 prior to the addition of e.

Flag Notation:

● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,

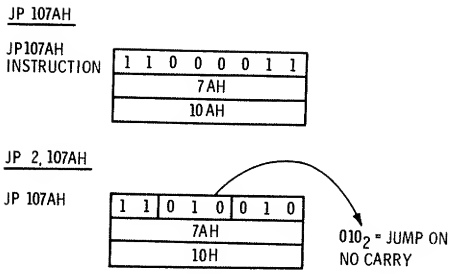


Fig. 5-20. JP and JPCC instruction examples.

not change the condition codes, while RES resets the indicated bit and does not change the condition codes. Fig. 5-19 shows the three kinds of bit instructions and examples of their use with various addressing modes.

JUMP GROUP

The instructions in the jump group are shown in Table 5-9. Basically, these can be divided into jumps, calls, and returns. *Jumps* cause a transfer to another location in memory and do not save the contents of the program counter to mark where the jump occurred. *Calls* perform the same action as a jump, but save the PC in the memory stack so that return may be made to the instruction following the call. *Returns* effect the transfer back to the instruction following the call by *popping* the stack and restoring the contents of the top of stack to the program counter. Calls and returns are used for subroutine processing. Subroutines are segments of code ranging from several instructions to hundreds of instructions that are called from many parts of a program. This avoids redundancy in writing the subroutine code many times throughout the program and saves memory and development time.

Two of the jump instructions JP NN and JP CC,NN exist in the 8080 and 8008 in extended addressing and are shown in Fig. 5-20. The NN field is the jump address. JP NN jumps *unconditionally* to the address, while JP CC,NN jumps to the address if the *conditions* CC are met. The encoding of the CC field is as follows:

CC	Condition	
000	Z = 0	(nonzero)
001	Z = 1	(zero)
010	C = 0	(no carry)
011	C = 1	(carry)
100	P = 0	(parity odd)
101	P = 1	(parity even)
110	S = 0	(positive)
111	S = 1	(negative)

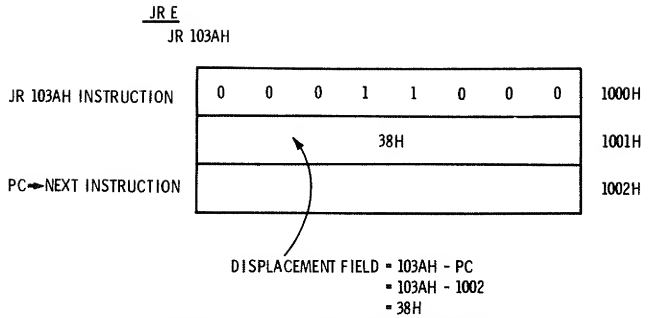


Fig. 5-21. JR E instruction example.

In addition to extended addessing, the Z-80 allows register indirect HL and indexed addressing for the JP NN instruction.

The remaining jumps are all of the relative addressing kind. JR E emulates the former jump. JR E is an unconditional relative jump to the effective address and is shown in Fig. 5-21. JR C,E; JR NC,E; JR Z,E; and JR NZ,E are relative conditional jumps that perform the jump if the carry is set or reset or if the zero flag is set or reset, respectively. The DJNZ E instruction is unique in that it decrements the contents of the B register. If the result is nonzero, the jump is performed; if zero, the next instruction in sequence is executed.

The two call instructions in this group also appear in the 8080 and 8008. CALL NN is an unconditional call and CALL CC,NN conditionally calls the subroutine at address NN. The conditions CC are the same as in the previous list. Likewise, RET and RET CC are also identical to the 8080 and 8008 instructions. RET unconditionally returns to the instruction after the call, while RET CC conditionally returns based on the CC field and the state of the condition-code register.

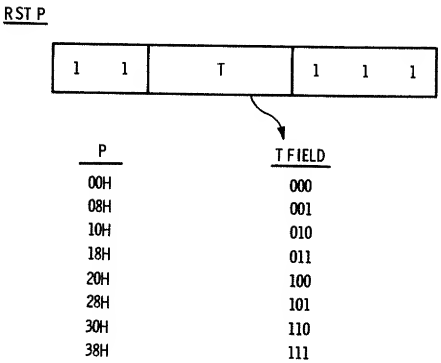


Fig. 5-22. RST P instruction.

Table 5-10. Input and Output Group

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
IN A,(n)	A ← (n)	●	●	●	●	●	●	11 011 011 ← n →	2	3	10	n to A_0–A_7 Acc to A_8–A_{15}
IN r,(C)	r ← (C) if r = 110 only the flags will be affected	●	↕	P	↕	0	↕	11 101 101 01 r 000	2	3	11	C to A_0–A_7 B to A_8–A_{15}
INI	(HL) ← (C) B ← B − 1 HL ← HL + 1	●	① ↕	X	X	1	X	11 101 101 10 100 010	2	4	15	C to A_0–A_7 B to A_8–A_{15}
INIR	(HL) ← (C) B ← B − 1 HL ← HL + 1 Repeat until B = 0	●	1	X	X	1	X	11 101 101 10 110 010	2 2	5 (If B ≠ 0) 4 (If B = 0)	20 15	C to A_0–A_7 B to A_8–A_{15}
IND	(HL) ← (C) B ← B − 1 HL ← HL − 1	●	① ↕	X	X	1	X	11 101 101 10 101 010	2	4	15	C to A_0–A_7 B to A_8–A_{15}
INDR	(HL) ← (C) B ← B − 1 HL ← HL − 1 Repeat until B = 0	●	1	X	X	1	X	11 101 101 10 111 010	2 2	5 (If B ≠ 0) 4 (If B = 0)	20 15	C to A_0–A_7 B to A_8–A_{15}
OUT (n),A	(n) ← A	●	●	●	●	●	●	11 010 011	2	3	11	n to A_0–A_7 Acc to A_8–A_{15}

OUT (C),r	(C) ← r	●	●	●	●	●	●	11 101 101 01 r 001	2	3	12	C to A_0–A_7 B to A_8–A_{15}
OUTI	(C) ← (HL) B ← B — 1 HL ← HL + 1	●	① ↕	X	X	1	X	11 101 101 10 100 011	2	4	15	C to A_0–A_7 B to A_8–A_{15}
OTIR	(C) ← (HL) B ← B — 1 HL ← HL + 1 Repeat until B = 0	●	1	X	X	1	X	11 101 101 10 110 011	2 2	5 (If B ≠ 0) 4 (If B = 0)	20 15	C to A_0–A_7 B to A_8–A_{15}
OUTD	(C) ← (HL) B ← B — 1 HL ← HL — 1	●	① ↕	X	X	1	X	11 101 101 10 101 011	2	4	15	C to A_0–A_7 B to A_8–A_{15}
OTDR	(C) ← (HL) B ← B — 1 HL ← HL — 1 Repeat until B = 0	●	1	X	X	1	X	11 101 101 10 111 011	2 2	5 (If B ≠ 0) 4 (If B = 0)	20 15	C to A_0–A_7 B to A_8–A_{15}

Notes:
① If the result of B — 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation:
● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

RETI and RETN are two special instructions that provide for special actions for returning from an external maskable interrupt (RETI) and nonmaskable interrupt (RETN). They will be discussed in Chapter 7.

RST P is also an instruction present in the 8080 and 8008. It is used for two operations. The primary operation is as an instruction that an interrupting device "jams" onto the data bus to effect a vectored interrupt. The subordinate function is to allow a special call to one of eight page 0 locations. The interrupt functions will be discussed in Chapter 7. When the RST P is used to call a page 0 location, the instruction acts as any unconditional call. The jump is made to one of eight page 0 locations based on the T field of the RST P as shown in Fig. 5-22.

INPUT AND OUTPUT GROUP

The last grouping of Z-80 instructions (Table 5-10) is the Input and Output group. The instructions in this group allow for transfer of 8-bit bytes of data to and from CPU registers A, B, C, D, E, H, or L with any of 256 possible I/O device addresses specified in the instruction. In addition, block transfers similar to the block transfers in the previous block transfer group can be implemented. Up to 256 bytes may be transferred semi-automatically or automatically between an I/O device and a memory block by using the I/O block transfer instructions (INI, INIR, IND, INDR, OUTI, OTIR, OUTD, and OTDR). The I/O instructions will be covered in detail in Chapter 15 of this section.

CHAPTER 6

Flags and Arithmetic Operations

The Z-80 flags have been briefly mentioned in previous chapters. This chapter discusses the flags in detail and the operations in the Z-80 which affect them. The flag register format is shown in Fig. 6-1. Although the flags exist as individual flip-flops within the CPU, they are logically grouped to simplify saving and restoring the flags for interrupts and other functions.

The Z flag, S flag, CY flag, and parity (overflow flag) may be tested by the conditional jumps described in Chapter 5. The conditional jump effectively tests the results of arithmetic, logical, shift, I/O or other operations preceding the conditional jump. The H and N flags are used to facilitate decimal or (bcd) arithmetic operations.

Z FLAG

The Z flag (bit position 6) is set if the result of certain instruction executions was zero. The Z flag will be set if the result is zero and reset if the result is nonzero for the instructions shown in Table 6-1.

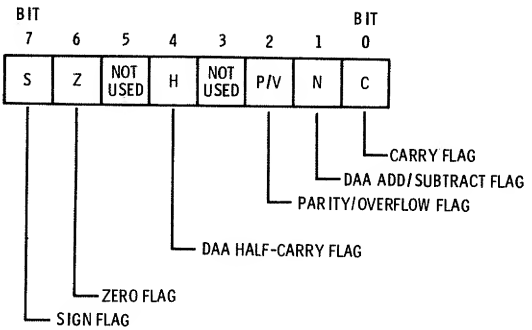


Fig. 6-1. Flag register format.

Table 6-1. Zero Flag Actions

Group	Instruction	Action
8-Bit Load Group	LD A,I LD A,R	Set Z if I register=0, otherwise reset Z Set Z if R register=0, otherwise reset Z
Search Group	CPI, CPIR, CPD,CPDR	Set if A=(HL), otherwise reset
8-Bit Arithmetic Group	ADD A,S SUB S OR S XOR S CP S INC S DEC S	Set if result=0, otherwise reset
General-Purpose Arithmetic Group	DAA NEG	
16-Bit Arithmetic Group	ADC HL,SS SBC HL,SS	
Rotate and Shift Group	RLC S RL RRC S RR S SLA S SRA S SRL S RLD RRD	
Bit Test Group	BIT B,S	Set if designated bit=0, otherwise reset
Input and Output Group	INR,(C) INI,IND, INIR,INDR, OUTI,OUTD OTIR,OTDR	Set if input data=0, otherwise reset Set if B — 1 = 0, otherwise reset Set Set if B — 1 = 0, otherwise reset Set

As the table shows, the Z flag is affected principally for arithmetic, logical, and shift operations. Loads and stores have no effect on the Z flag except for the two cases of LD A,I and LD A,R. The search group is essentially a comparison or subtraction and the Z flag is also affected here. The bit test group is effectively a logical AND and the Z flag is again set or reset on the result. Note that *except for the instructions shown,* no other instructions have an effect on the Z flag. Once the Z flag is set or reset by an ADD A,S, for example, it will not be reset until the next instruction in this group is encountered. This is an important point as it means that the conditional jumps on the Z flag, JP Z,NN; JP NZ,NN; JR Z,E; and JR NZ,E,

do not have to be immediately executed after the instruction that affects the Z flag. As long as no other instructions in Table 6-1 appear before the test, the conditional jump may be deferred as long as desired. In actual practice, the conditional jump will occur close to the instruction setting the flag, however.

The Z flag would normally be tested for a variety of conditions. Some of the more common ones are shown below:

1. Equality of two operands after a CP (compare).
2. Increment or decrement of an index count down to 0.
3. Bit test result of 0.
4. Result after a shift of 0, signifying no additional data in operand.
5. Zero field after AND.

SIGN FLAG

The S flag (bit-position 7) is set if the result of certain instruction executions are negative and reset if they are positive. Since in two's complement notation, positive quantities have bit $7 = 0$ and negative quantities have bit $7 = 1$, the sign flag reflects the true sign of the result. The S flag is affected by the instructions shown in Table 6-2.

Sign flag actions are very similar to zero flag actions as shown in the table. The sign flag is primarily affected by arithmetic and shift operations, including the comparisons in the search group. Note that for some instructions the flag *is affected,* but that the state is not known. The same ground rules on testing of the sign flag apply as for testing of the zero flag; the conditional branch must be performed before an instruction is executed that affects the flag. Some of the common conditions for which the sign flag is tested are:

1. Comparisons of two operands (greater, less than, etc.)
2. Increment or decrement of an index count past 0
3. Shift of a 1 (or 0) bit into sign bit position

CARRY FLAG

While the zero and sign flag were associated with arithmetic, shift, and logical operations, the carry flag is associated principally with arithmetic and shift operations as shown in Table 6-3, although it is reset by the logical instructions.

The carry flag is used to:

1. Test the results of the comparison of two operands
2. Test the results of a shift operation

3. Provide a means to do multiple-precision arithmetic

When a carry is tested after a compare of two *unsigned* operands, the carry will be set if in the comparison OP1 : OP2 (OP1 − OP2), OP1 is greater or equal to the second operand OP2. Some examples of this are shown in Fig. 6-2. The comparison could also have been tested by the sign bit, which is a more common way to implement

Table 6-2. Sign Flag Actions

Group	Instruction	Action
8-Bit Load Group	LD A,I	Set if I register is negative, otherwise reset
	LD A,R	Set if R register is negative, otherwise reset
Search Group	CPI,CPIR, CPD,CPDR	Set if result is negative, otherwise reset
8-Bit Arithmetic Group	ADD A,S ADC A,S SUB S SBC A,S AND S OR S XOR S CP S INC S DEC S	
General-Purpose Arithmetic Group	DAA	Set if msb of A = 1, otherwise reset
	NEG	Set if result is negative, otherwise reset
16-Bit Arithmetic Group	ADC HL,SS	
	SBC HL,SS	
Rotate and Shift Group	RLC S RL S RRC S RR S SLA S SRA S SRL S	
	RLD RRD	Set if A is negative after shift, reset otherwise
Bit Test Group	BIT B,S	Unknown
Input and Output Group	IN R,(C)	Set if input data is negative, otherwise reset
	INI,INIR, IND,INDR, OUTI,OTIR, OUTD,OTDR	Unknown

Table 6-3. Carry Flag Actions

Group	Instruction	Action
8-Bit Arithmetic Group	ADD A,S ADC A,S	Set if carry from bit 7, otherwise reset
	SUB S SBC S	Set if no borrow, otherwise reset
	AND S OR S XOR S	Reset
	CP S	Set if no borrow, otherwise reset
General-Purpose Arithmetic Group	DAA	Set if bcd carry, otherwise reset
	NEG	Set if A was not OOH before negate, otherwise reset
	CCF	Set if CY was 0 before CCF, otherwise reset
	SCF	Set
16-Bit Arithmetic Group	ADD HL,SS ADC HL,SS	Set if carry from bit 15, otherwise reset
	SBC HL,SS	Set if no borrow, otherwise reset
	ADD IX,PP ADD IY,RR	Set if carry from bit 15, otherwise reset
Rotate and Shift Group	RLCA RLA	Set from A bit 7
	RRCA RRA	Set from A bit 0
	RLC S RL S	Set from bit 7 of operand
	RRC S RR S	Set from bit 0 of operand
	SLA S	Set from bit 7 of operand
	SRA S	Set from bit 0 of operand
	SRL S	Set from bit 0 of operand

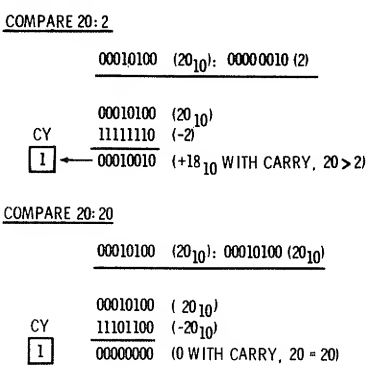


Fig. 6-2. Carry comparisons.

the test. In the shift instructions, the carry is set or reset by the state of the bit shifted out of the operand and this provides a convenient way of testing and conditionally branching on a carry (1 bit) or no carry (0 bit). Finally, the carry is set from the high order bit of the result during multiple-precision adds or subtracts. The first add is an ADD (without carry) while successive adds of higher-order operands are ADC types, which add in the carry from the lower-order result (see Fig. 6-3).

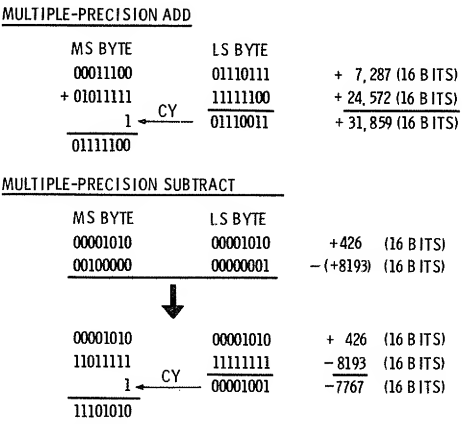


Fig. 6-3. Carry in multiple-precision operations.

PARITY/OVERFLOW FLAG

The parity/overflow flag (bit-position 2 in the flag register) is a dual-purpose flag. In the parity case, the flag is set to represent odd parity of the result of the operation. Odd parity occurs when the sum of the eight bits of the result is even. In this case, the parity bit is set. If the sum is odd, the parity bit is reset. (See Fig. 6-4). When the P/V flag is used to represent overflow, the flag is set if arithmetic overflow occurs after an arithmetic operation. Arithmetic overflow will occur if in an add or subtract operation of two numbers of like signs the sign of the result changes, indicating that the result is too large to be held in eight (or sixteen) bits. Examples of overflow conditions are shown in Fig. 6-5. Table 6-4 lists the instructions that affect the parity/overflow flag.

RESULT = 00101011

4 ONE BITS = EVEN PARITY, SET P/V FLAG

Fig. 6-4. P/V flag used as parity indicator.

RESULT = 00111110

5 ONE BITS = ODD PARITY, RESET P/V FLAG

Table 6-4. Parity/Overflow Flag Actions

Group	Instruction	Action
8-Bit Load Group	LD A,I LD A,R	Contents of IFF2
Block Transfer and Search Group	LDI,LDD, CPI,CPIR, CPD,CPDR	Set if BC — 1 $\neq$ 0, otherwise reset
	LDIR,LDDR	Reset
8-Bit Arithmetic Group	ADD A,S ADC A,S SUB S SBC A,S	Set if overflow, otherwise reset
	AND S OR S XOR S	Set if parity even, otherwise reset
	CP S	Set if overflow, otherwise reset
	INC S	Set if operand was 7FH before increment, otherwise reset
	DEC S	Set if operand was 80H before increment, otherwise reset
General-Purpose Arithmetic Group	DAA	Set if (A) parity even, otherwise reset
	NEG	Set if (A) was 80H before negate, otherwise reset
16-Bit Arithmetic Group	ADC HL,SS SBC HL,SS	Set if overflow, otherwise reset
Rotate and Shift Group	RLC S RL S RRC S RR S SLA S SRA S SRL S RLD S RRD S	Set if parity even, otherwise reset
Bit Test Group	BIT B,S	Unknown
Input and Output Group	IN R,(C)	Set if parity even, otherwise reset
	INI,INIR, IND,INDR, OUTI,OTIR, OUTD,OTDR	Unknown

Table 6-5. H and N Flag Actions

Group	Instruction	H Action	N Action
8-Bit Load Group	LD A,I LD A,R	Reset	Reset
Block Transfer and Search Group	LDI,LDIR, LDD,LDDR		
	CPI,CPIR,CPD, CPDR	Set if no borrow from bit 4, otherwise reset	Set
8-Bit Arithmetic Group	ADD A,S ADC A,S	Set if no carry from bit 3, otherwise reset	Reset
	SUB S SBC A,S	Set if no borrow from bit 4, otherwise reset	Set
	AND S OR S XOR S	Set	Reset
	CP S	Set if no borrow from bit 4, otherwise reset	Set
	INC S	Set if carry from bit 3, otherwise reset	Reset
	DEC S	Set if no borrow from bit 4, otherwise reset	Set
General-Purpose Arithmetic Group	DAA	Indeterminate	Not affected
	CPL	Set	Set
	NEG	Set if no borrow from bit 4, otherwise reset	Set
	CCF	Not affected	Reset
	SCF	Reset	Set
16-Bit Arithmetic Group	ADD HL,SS ADC HL,SS	Set if carry out of bit 11, otherwise reset	Reset
	SBC HL,SS	Set if no borrow from bit 12, otherwise reset	Set
	ADD IX,PP ADD IY,RR	Set if carry out of bit 11, otherwise reset	Reset
Rotate and Shift Group	RLCA RLA RRCA RRA RLC S RL S RRC S RR S SLA S SRA S SRL S RLD RRD	Reset	Reset

Table 6-5. H and N Flag Actions—cont

Group	Instruction	H Action	N Action
Bit Test Group	BIT B,S	Set	Reset
Input and Output Group	INR,(C)	Reset	Reset
	INI,INIR, IND,INDR, OUTI,OTIR, OUTD,OTDR	Indeterminate	Set

```
                01111111    +127(10)
P/V             +01000000   + 64(10)
[1]             10111111    +191(10)  (OVERFLOW, TOO LARGE TO HOLD IN 8 BITS)

                10000011    -125(10)
P/V             +10000010   -126(10)
[1]             00000101    -251(10)  (OVERFLOW, TOO LARGE TO HOLD IN 8 BITS)

                00100000    +32(10)
P/V             00100000    +32(10)
[0]             01000000    +64(10)   (NO OVERFLOW)
```

Fig. 6-5. Overflow conditions and P/V flag.

H AND N FLAGS

The H and N flags (bit-positions 4 and 1, respectively) are two flags that cannot be tested by conditional jump instructions. They are used by the Z-80 CPU for bcd arithmetic operations. H represents the half-carry from the four least significant bits of the result (least significant bcd digit) and N is the subtract flag, which is set to indicate whether an add or subtract was last executed. Table 6-5 shows the instructions affecting the H and N flags.

Note that in the general case, an add instruction resets the N flag and a subtract sets the N flag. This is also true for increments (adds) and decrements (subtracts). When the DAA instruction is executed after an add or subtract, it senses the N flag and half-carry H flag and properly adjusts the result from a binary to bcd result. For an add ($N = 0$), a binary result must be corrected by adding a six to the bcd digit position under certain conditions. Those conditions are:

1. If there has been a carry from the bcd digit ($H = 1$ or $C = 1$).
2. If there was no carry but bcd digit position has a value greater than 1001_2.

ADD 11 AND 22 IN BCD

```
0001 | 0001   (11 BCD)
0010 | 0010   (22 BCD)
0011 | 0011   (33 BCD)  CY = 0, H = 0
```

ADD 19 AND 29 IN BCD

```
0001 | 1001   (19 BCD)
0010 | 1001   (29 BCD)
0100 | 0010   (42 BCD! WRONG) CY = 0, H = 1
0000 | 0110   ADJUST BY +6 TO LOW ORDER BCD DIGIT
0100 | 1000   (48 BCD CORRECT)
```

ADD 91 AND 92 IN BCD

```
     1001 | 0001   (91 BCD)
CY   1001 | 0010   (92 BCD)
[1]  0010 | 0011   (23 BCD! WRONG) CY = 1, H = 0
CY   0110 | 0000   ADJUST BY +6 TO HIGH ORDER BCD DIGIT
[1]  1000 | 0011   (83 BCD WITH CY = 1 CORRECT)
```

ADD 99 AND 99 IN BCD

```
     1001 | 1001   (99 BCD)
CY   1001 | 1001   (99 BCD)
[1]  0011 | 0010   (32 BCD! WRONG) CY = 1, H = 1
CY   0110 | 0110   ADJUST BY +6 TO BOTH BCD DIGITS
[1]  1001 | 1000   (98 BCD WITH CY = 1 CORRECT)
```

Fig. 6-6. Bcd addition and use of CY and H.

Some examples of the above are shown in Fig. 6-6. For a subtract ($N = 1$), a binary result must be corrected by subtracting a six from a bcd digit position under certain conditions. If there is a half-carry, a six is subtracted from the least significant bcd digit position. If there is a carry, a six is subtracted from the most significant bcd digit position. If there are both a carry and half-carry, a six is subtracted from each bcd digit position. Fig. 6-7 illustrates the conditions for bcd subtract corrections.

Multiple-precision bcd arithmetic is easily possible by maintaining the carry from the last bcd addition or subtraction.

SUBTRACT 11 FROM 99 IN BCD

	1001 | 1001	(99 BCD)
	-0001 | 0001	(11 BCD)
	1000 | 1000	(88 BCD) CY = 0, H = 0

SUBTRACT 19 FROM 91 IN BCD

	1001 | 0001	(91 BCD)
CY	-0001 | 1001	(19 BCD)
[0]	0111 | 1000	(78 BCD, WRONG) CY = 0, H = 1
CY	0000 | 0110	ADJUST BY -6 TO LOW ORDER BCD DIGIT
[0]	0111 | 0010	(72 BCD WITH CY = 0, CORRECT)

SUBTRACT 91 FROM 19 IN BCD

	0001 | 1001	(19 BCD)
CY	-1001 | 0001	(91 BCD)
[1]	1000 | 1000	(88 BCD, WRONG) CY = 1, H = 0
CY	-0110 | 0000	ADJUST BY -6 TO HIGH ORDER BCD DIGIT
[1]	0010 | 1000	(28 BCD WITH CY = 0, CORRECT)

SUBTRACT 99 FROM 11 IN BCD

	0001 | 0001	(11 BCD)
CY	-1001 | 1001	(99 BCD)
[1]	0111 | 1000	(78 BCD, WRONG) CY = 1, H = 1
CY	0110 | 0110	ADJUST BY -6 TO BOTH BCD DIGITS
[1]	0001 | 0010	(12 BCD WITH CY = 0, CORRECT)

Fig. 6-7. Bcd subtraction and use of CY and H.

CHAPTER 7

Interrupt Sequences

Interrupts in the Z-80 serve the same purposes as interrupts in other microprocessors and computers—they signal the microprocessor that an external event has occurred that requires attention. Many times the external event is associated with the transfer of I/O data to and from the microcomputer timing functions, or abnormal or catastrophic external conditions.

When interrupts are associated with transfer of I/O data, the interrupt is a mechanism to overlap CPU processing time with the I/O activity. As an example of this kind of interrupt, let us assume that a microcomputer system is connected to a "high-speed" paper-tape reader. The paper-tape reader may be able to read data at the rate of 500 *frames*, or bytes, per second. Each new data byte will be available every 1/500 second or 2 milliseconds. If the program that reads data from the paper-tape reader is implemented without interrupts, it will read a byte of data by an IN instruction every 2 milliseconds and the entire read operation will take approximately 2.5 microseconds, as shown in Fig. 7-1. For the remainder of the time the program is simply continually querying the paper-tape reader (by means of another IN instruction to read *status* information) whether the next byte is available. If 500 bytes are to be read, and if the average CPU instruction time is 2.5 microseconds, then $(1/2.5 \times 10^{-6} - 500) = 399{,}500$ instruction times are lost while the CPU is idle awaiting the next byte of data.

Interrupts allow the CPU to make use of the idle time associated with I/O activity. With proper use of interrupts, the CPU may execute another portion of the program while the I/O idle time occurs and be informed of the availability of the next data byte by inter-

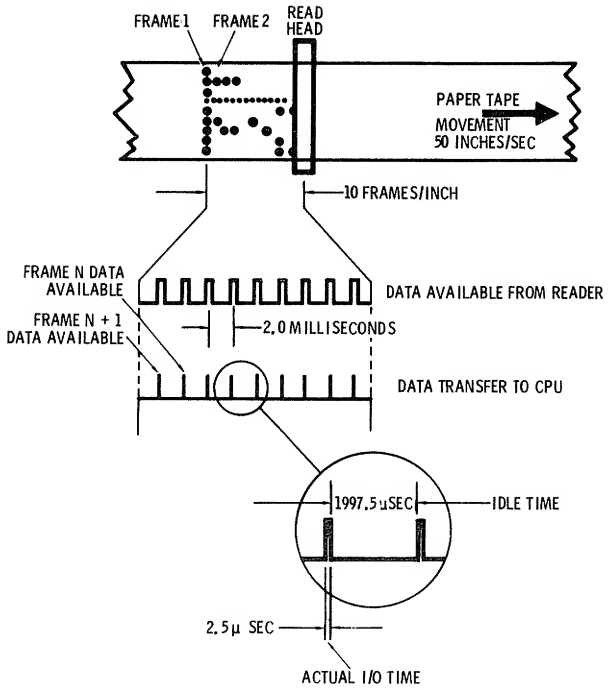


Fig. 7-1. I/O idle time example.

rupt action. An interrupt sequence will then be entered and the CPU can quickly pick up the next I/O data, do some minimal processing, and *exit* the interrupt sequence to return to the *interrupted* program. If multiple I/O activity is required, many I/O devices can operate in this fashion, signaling the CPU by *vectored interrupts* which device requires I/O attention. The Z-80 expands the 8-vectored interrupt capability of the 8080 to 128 separate vectored interrupts that are usable for I/O or other types of functions.

A second use of interrupts is to provide CPU timing functions. It is convenient to provide measured time intervals to the CPU to enable the CPU to maintain a *real-time clock* for system *time-out* functions or *time-of-day* indications. Typically, the time interval is provided via interrupts, with a *programmable counter-timer* interface that interrupts the CPU every tenth of a second, or so. The CPU will recognize the interrupt as a timer interrupt, enter the proper software interrupt processing routine, and adjust a system clock and/or perform other timing functions, and exit the interrupt routine to continue processing at the interrupted point.

A third use of interrupts is to signal abnormal or catastrophic system conditions. Typical conditions of which the CPU would be

informed is pending power failure or failure of a redundant portion of the computer system in a real-time system. Often these functions are implemented using a *nonmaskable* interrupt, since the interrupt must be recognized immediately and not deferred until current processing is completed. The Z-80 has provision for this kind of interrupt with a special NMI (nonmaskable interrupt) input that produces a separate interrupt action from other external interrupts.

Z-80 INTERRUPT INPUTS

As Fig. 3-1 shows, there are two interrupt inputs to the Z-80 microprocessor chip, the $\overline{\text{NMI}}$, nonmaskable interrupt, and the $\overline{\text{INT}}$, or normal external interrupt. The $\overline{\text{NMI}}$ input allows a single NMI interrupt while the INT allows up to 128 separate *vectored* interrupts by means of encoding from external device controllers or interrupt logic. The NMI is always *recognized* by the CPU. If the $\overline{\text{NMI}}$ becomes active, the automatic NMI actions are unconditionally implemented. The $\overline{\text{INT}}$ is recognized by the CPU only if an *interrupt enable* condition is present in the CPU. The interrupt enable is provided by a programmable flip-flop that can be set (interrupts enabled) or reset by the EI or DI instructions. The INT interrupt action is more complicated than the NMI action, since an external device must provide encoded data relating to the identification of the interrupting device. In addition, there are three different interrupt mode for the maskable INT interrupt modes 0, 1, and 2, that are set by instructions IM 0, IM 1, IM 2. Each mode provides a different interrupt action.

NMI INTERRUPT

When an NMI interrupt occurs ($\overline{\text{NMI}}$ goes low to active state), the interrupt is recognized at the end of the current instruction. The CPU then effectively performs a Restart instruction to location 0066H. As Chapter 5 describes, a Restart pushes the current contents of the PC into the stack, and transfers control to one of eight locations 0000, 0008H, . . . 0038H. The NMI action is the same as a Restart, but the transfer address is always 0066H. As an example of the stack actions during the NMI, let us assume that the CPU was executing the instructions shown in Fig. 7-2. The NMI interrupt occurs during the RRCA instruction. At the end of the RRCA, the contents of the program counter is 102BH. As the NMI interrupt is implemented by the CPU, the contents of the PC is pushed into the stack as shown, and the stack pointer decremented by two. The instruction at location 0066H is then executed.

We will assume in this case that the external condition causing the interrupt is not a catastrophic one, and the system will remain operative. Since the flag register and all CPU registers must be restored exactly as they were at the time the interrupt occurred when a return to location 102AH is made, the routine at 0066H must somehow *save the environment.* The easiest way to do this in the Z-80 is simply to switch to the alternate set of registers by two exchange

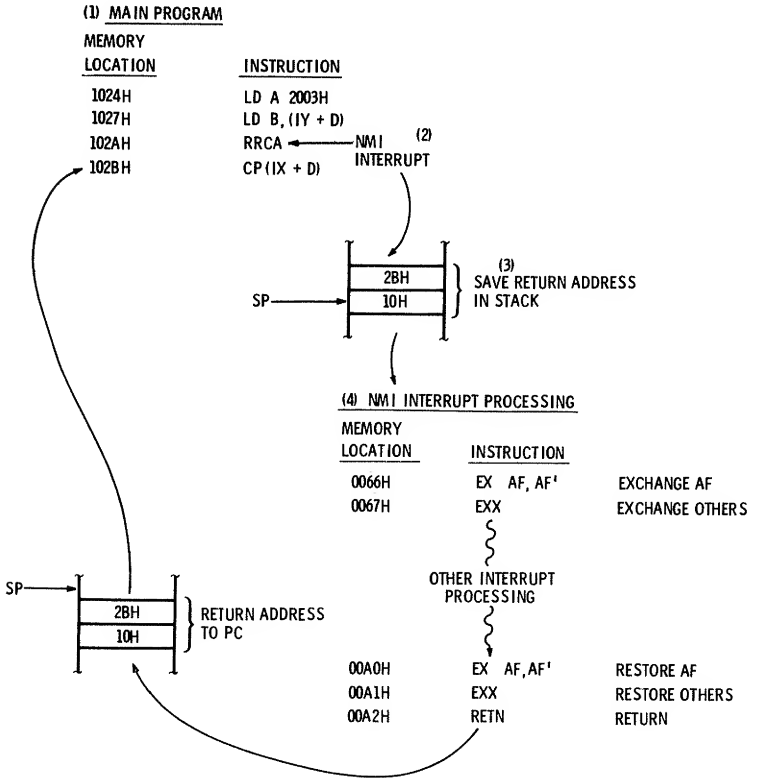


Fig. 7-2. NMI interrupt processing.

instructions. Interrupt processing now proceeds for the NMI condition. The size of the interrupt processing routine is dependent on the amount of processing to be performed. At the end of processing, two exchange instructions restore the CPU registers and flags to their status at the time of the interrupt and a special instruction RETN is executed to return from the NMI interrupt. RETN pops the program counter from the stack and causes the CPU to start execution of the next instruction at 102AH. At this point, all CPU registers and flags appear as if the NMI interrupt had never occurred.

There is a subtlety about the NMI that has not been mentioned previously. There are actually two interrupt flip-flops in the Z-80 CPU, designated IFF_1 and IFF_2. IFF_1 is the flip-flop associated with disabling or enabling the maskable interrupt. IFF_2 is used to temporarily store the state of IFF_1 when an NMI interrupt occurs. In addition to storage of IFF_1, the NMI resets IFF_1 *so that no maskable external interrupt can occur.* This avoids the *reentrancy* problems of simultaneous NMI and maskable interrupts. When the RETN is executed, the previous state of IFF_1 is restored by transferring the contents of IFF_2. The maskable interrupt status (enabled or disabled) is now the same as before the NMI interrupt. If the program can allow an external maskable interrupt to occur during the time an NMI interrupt is being processed, an EI instruction can be executed after storage of the registers and flags. External interrupts would then be enabled during NMI processing time, although this action would probably not be typical in most applications.

MASKABLE INTERRUPT MODE 0

Interrupt mode 0 is the default CPU interrupt mode on start up. When signal $\overline{\text{RESET}}$ initially becomes active, mode 0 is set in the CPU. Mode 0 may also be set by execution of an IM 0 instruction. Interrupt mode 0 is identical to the interrupt processing in the 8080. If mode 0 is set and the interrupt enable flip-flop IFF_1 is set and an external maskable interrupt occurs, the following actions take place:

1. Interrupt occurs ($\overline{\text{INT}}$ goes low)
2. At end of current instruction, CPU recognizes interrupt
3. CPU responds by $\overline{\text{IORQ}}$ and $\overline{\text{M1}}$ signal
4. External device recognizes the $\overline{\text{IORQ}}$ and $\overline{\text{M1}}$ response and outputs a Restart instruction to data bus encoded with 000_2 to 111_2 as T field
5. CPU strobes in Restart and executes the instruction causing transfer to page 0 location corresponding to T field (0, 8, 10 . . . , 38H)
6. Instructions defining the interrupt processing routine are executed
7. An RETI instruction is executed returning control to next instruction after interrupt

Mode 0 interrupt processing is similar to the NMI interrupt processing. Both execute a Restart, both transfer to a page 0 location, and both require an RET instruction at the end of the interrupt processing. Let us illustrate a maskable interrupt mode 0 by hy-

pothesizing a paper-tape–reader controller with interrupt capability. Fig. 7-3 shows the interrupt action. When the next frame of tape has been read, the paper-tape controller brings the $\overline{INT}$ line low. When the interrupt is recognized by the CPU, lines $\overline{IORQ}$ and $\overline{M1}$ are brought low (not to scale in figure). This is decoded by the controller as an interrupt acknowledge and the controller jams an RST 20H instruction onto the data bus. The CPU executes the RST 20H, pushing the contents of the program counter (3332H) into the stack and transferring control to page 0 location 20H. At 20H a JP FEE0H is executed to transfer control to the paper-tape interrupt

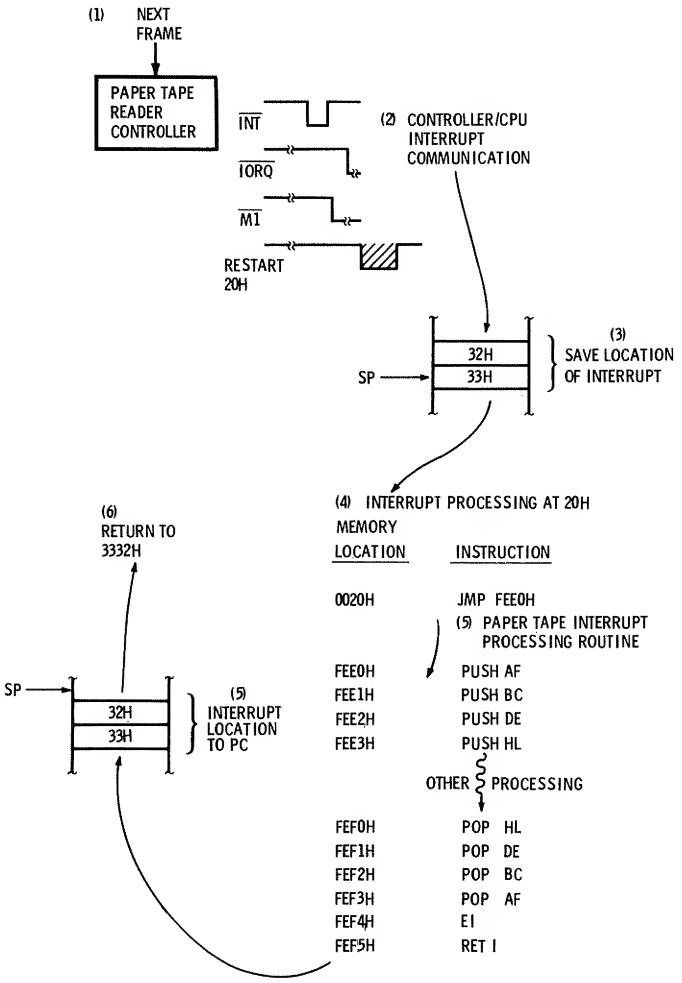


Fig. 7-3. Mode 0 interrupt processing.

processing routine at FEE0H. (The interrupts have automatically been disabled on the receipt of the maskable interrupt and will not be reenabled until an EI instruction is executed.) The paper tape saves the contents of the registers and flags by a series of pushes into the stack (an alternative way to save the environment from register switching). Instructions are then executed to read in the character, reset the interrupt states in the paper-tape controller, and, in general, process the data. At the end of the interrupt processing routine, the environment is restored by a series of pops in reverse order, an EI instruction is executed for the next character interrupt, and an RETI returns control to location 3332H.

The above example considers only one interrupting device, the paper-tape–reader controller. It is possible to have many interrupting devices in this mode and mode 2 of the Z-80 interrupt sequence. When many devices are capable of interrupting, some means of *prioritizing* the devices must be implemented to avoid simultaneous interrupt requests from two or more devices over the same interrupt line. If a prioritizing scheme is not used, confusion will result as each device thinks that it has received an interrupt acknowledge.

In a prioritizing scheme, each device is assigned a priority from high to low as shown in Fig. 7-4. The eight devices shown here connect to a *priority interrupt control unit* (Intel 8214). The priority interrupt control unit and associated logic allows only one device to interrupt at a time and handles all interrupt communication between the CPU and interrupting devices in interrupt mode 0. If several devices have simultaneous interrupt requests, the control unit will determine the highest-priority request, bring down the $\overline{\text{INT}}$ line, and jam the proper Restart instruction onto the data bus

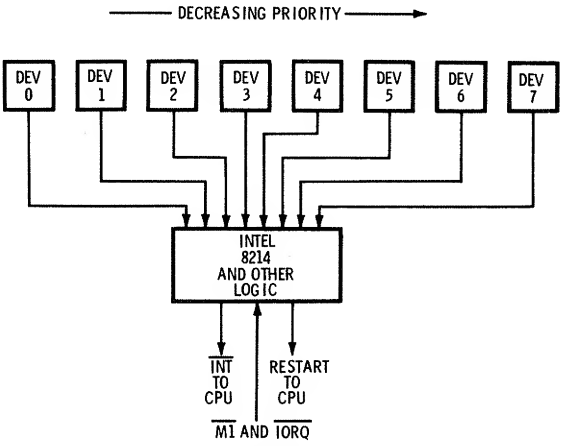


Fig. 7-4. Priority encoding for interrupt mode 0.

after interrupt acknowledge. At any time during the servicing of one interrupt, one or more higher-priority interrupts may become active. When this occurs, the interrupt sequences for the higher-level interrupts are entered. If the interrupt control flip-flop IFF_1 has been properly maintained to prevent interrupts from other devices at critical times, such as saving the environment, there should be no conflicts in servicing n number of interrupts in *nested* fashion. Further examples of prioritizing will be discussed for the mode 3 interrupt sequence.

MASKABLE INTERRUPT MODE 1

The next two interrupt modes, mode 1 and mode 2, are not compatible with the Intel 8080. Mode 1 is set by the IM 1 instruction. The interrupt actions of mode 1 are identical to the nonmaskable interrupt response, except that the Restart location is location 0038H rather than 0066H. If mode 1 is set and the maskable interrupts are enabled, then an interrupt request on $\overline{INT}$ will cause a Restart to location 0038H. The contents of the program counter will be saved in the stack and the interrupt servicing routine at location 0038H will be entered. The advantage of mode 1, as in the NMI interrupt, is that no external logic is required to jam the Restart onto the data bus at the proper time. An external interrupt can be implemented with only enough logic to bring $\overline{INT}$ active and recognize the interrupt acknowledge. Of course, only one interrupt level is permitted in this mode.

MASKABLE INTERRUPT MODE 2

The last and most powerful interrupt mode is interrupt mode 2. This mode allows up to 128 interrupts from external devices, each fully vectored to an interrupt location anywhere in memory. Furthermore the peripheral modules in the Zilog family, such as the Z-80 PIO (parallel I/O), Z80-SIO (serial I/O), and Z-80-CTC (counter-timer circuit) may easily be connected in *daisy-chained* fashion to allow for complete prioritizing of all interrupt levels.

Mode 2 is set by an IM 2 instruction. The heart of this interrupt mode is an interrupt vector table anywhere in memory. In general, the table is $(2 \times N)$ bytes long, where N is the number of interrupts in the system and the start of the table is pointed to by IIIIIIII-00000000_2, where I is the contents of the Interrupt Vector register I. For any interrupt, the I register supplies the eight most significant bits of the table address while the interrupting device supplies the eight least significant bits of the table address. The table has up to 128 entries as shown in Fig. 7-5. Each entry is two bytes

long and represents a memory address for the interrupt servicing routine for a particular device in standard 8008/8080/Z-80 order—most significant byte last.

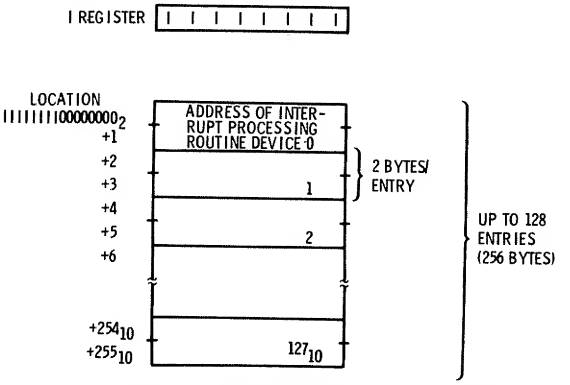


Fig. 7-5. Interrupt mode 2 interrupt vector table.

The general sequence for interrupt mode 2 is this:

1. If IM 2 is set and $IFF_1 = 1$ and $\overline{INT}$ is active, the CPU recognizes the interrupt at the next M1 cycle.
2. The interrupting device responds to the interrupt acknowledge with an 8-bit value.
3. The 8-bit value is merged into a memory address with the contents of the I register.
4. The CPU pushes the contents of the PC into the stack.
5. The contents of the interrupt vector table is accessed using the address computed in step 3.
6. The PC is loaded with the contents of the interrupt vector table entry to effectively cause a jump to the interrupt servicing routine defined by the address vector in the table.

Fig. 7-6 shows an example of this process. The interrupt vector table is located at F000H. The table has ten entries of two bytes each defining ten interrupt servicing routines for the ten interrupting devices. Note that two of the addresses are identical, indicating that the same interrupt service will be performed for two devices. The interrupt vector register I has previously been loaded with F0H. When an external interrupt is generated for device number 5 and the interrupt acknowledge is received, the device controller places an 8-bit vector on the data bus, in this case 02H. The 02H is merged with the I register to form address F002H. The CPU now reads the two bytes at locations F002H and F003H to find the address of the interrupt servicing routine, after pushing the current contents of the

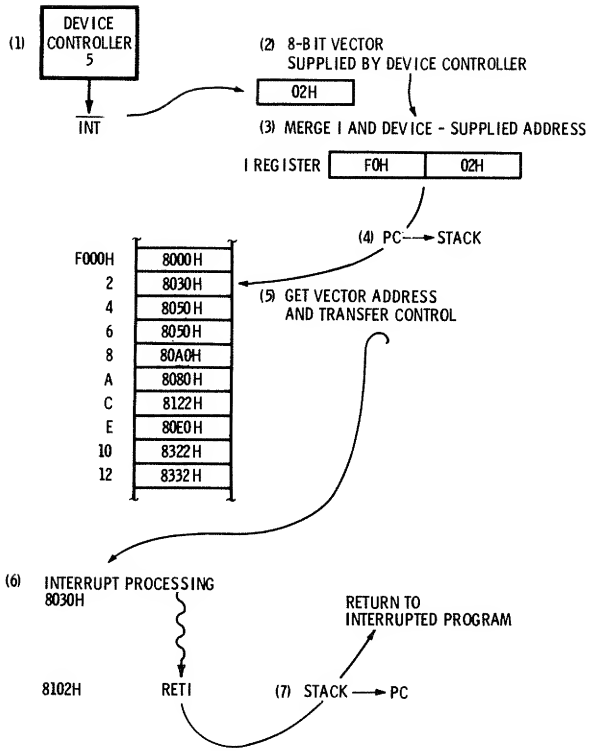


Fig. 7-6. Interrupt mode 2 example.

PC into the stack. The table entry is 8030H, and the CPU transfers control to this location for interrupt servicing. The interrupt servicing is then performed and an RETI is executed, terminating the interrupt action and returning control to the interrupted location as the PC is loaded with the return address from the stack.

Note that the interrupting device could supply any eight bits for the vector, not necessarily that address associated with its I/O device address in the execution of IN and OUT instructions, although it is convenient to have device address 0 associated with table entry F000, device 1 associated with table entry F002H, etc. Note also that the interrupting device really supplies only a 7-bit address. The least significant bit is always 0, since each table entry is 2 bytes long. Device number n would conveniently supply vector $2 \times N$, if the table were ordered in this fashion.

In a prioritizing scheme used by Z-80 peripheral devices, each device has an implicit priority as shown in Fig. 7-7. Here the devices are Zilog Z-80 PIO (Parallel I/O) modules. Each PIO has automatic interrupt prioritizing "built-in" and is specifically designed for inter-

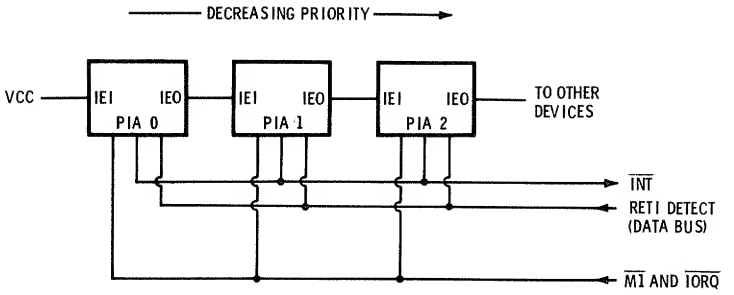


Fig. 7-7. Z-80 interrupt prioritizing interrupt mode 2.

rupt mode 2 operation. Every PIO is connnected to the Z-80 CPU $\overline{INT}$ line in a "wire-or" configuration ($\overline{INT}$ is directly connected, without *buffering* by gating). If the IEI (interrupt enable in) signal from higher priority devices is high (positive), no higher priority device has requested service and an interrupt request may be generated from the PIO. Prior to the interrupt request, the IEO (interrupt enable out) goes low, indicating to lower-priority devices that they may not request interrupt service by bringing down the $\overline{INT}$ line. When the interrupt acknowledge occurs, the PIO automatically jams the proper 8-bit mode 3 vector onto the data bus to vector the interrupt to the proper memory location. At the end of interrupt servicing, the RETI is detected by decoding the instruction op code and the interrupt for the current PIO is completed. IEO for the current PIO is brought high, enabling interrupts from lower-priority devices.

A prioritizing scheme such as the above not only handles the problem at simultaneity of interrupt requests, but also enables multilevels of interrupts. To illustrate the operation of nested interrupts let us

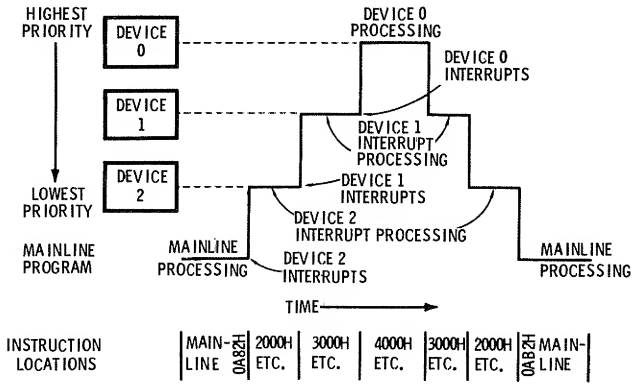


Fig. 7-8. Nested interrupt example.

use the configuration shown in Fig. 7-8. As before, the interrupt vector table is at location F000H and the interrupt vector register has previously been loaded with F000H. The interrupting devices are labeled 0, 1, and 2; and they have priority in that order. Interrupt mode 2 has been previously set. During execution of *main-line* program location 0A82H, device 2 interrupts. Interrupt processing routine 2000H is entered and the environment is saved as shown. After enabling interrupts, device number 1 interrupts the interrupt processing routine for device number 2, jumping to location 3000H. Finally, device 0 interrupts during the middle of the interrupt processing routine for device number 1, causing interrupt 0 processing routine at 4000H to be entered. This routine is completed by an RETI and the processing routine at 3000 is reentered. This routine is then completed and, after the RETI, the processing routine at 1000H is again reentered. Finally, the lowest level processing routine at 1000H is completed, an RETI executed, and a return mode to the main-line program at 0A82H. At one time during the sequence, three nested interrupts were involved. Assuming that the environment was properly saved and restored and the interrupts were disabled at proper times, no problems should have been encountered with this scheme, or even a great deal more complex interrupt structure.

CHAPTER 8

Interfacing Memory and I/O Devices to the Z-80

As the Z-80 requires only a single-phase clock and a single 5-volt power supply, a minimum Z-80 system can be implemented with few additional components. This chapter will describe simple interfacing cases of the Z-80 and ROM memory, static RAM memory, dynamic RAM memory, and the Zilog Z-80 PIO.

MINIMUM Z-80 SYSTEM

The components required for a minimum Z-80 system are:

1. a 5-volt power supply
2. a single-phase TTL-compatible clock
3. a means to reset (restart) the system
4. ROM or PROM memory to contain the program
5. I/O interfacing and devices
6. the Z-80 CPU

Fig. 8-1 shows a minimum system with the above components. A momentary switch resets the CPU and starts program execution at location 0 by bringing down the $\overline{\text{RESET}}$ signal to a logic 0. As the reader will recall from Chapter 3, the $\overline{\text{RESET}}$ signal disables interrupts, sets the I and R registers equal to 0, sets interrupt mode 0, and sets the program counter to 0. A simple timing circuit provides a square-wave clock input at 2.5 to 4.0 MHz. The clock runs continuously. The ROM memory is a fast-access (greater than 250 nanoseconds) 512 × 8 ROM addressable by lines A0 – A8 of the Z-80.

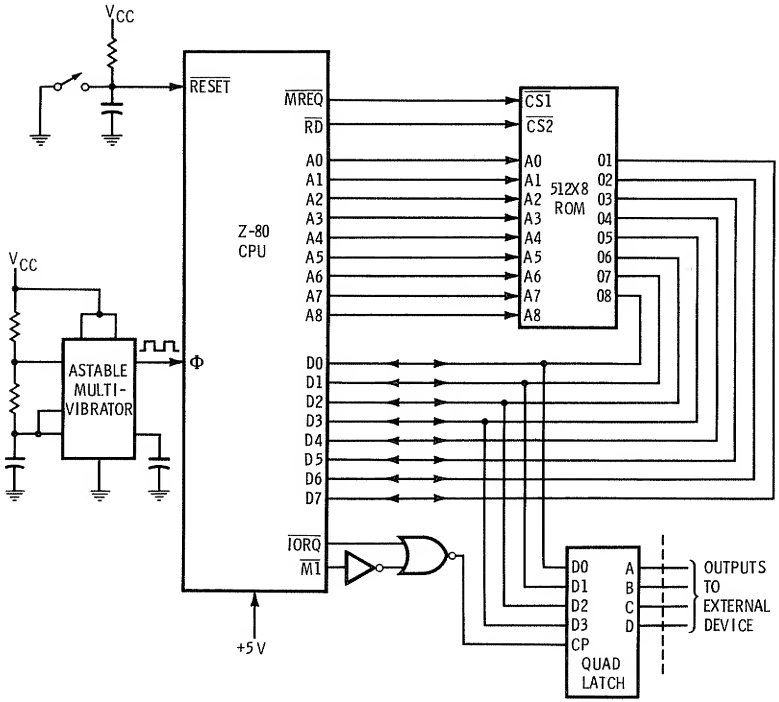


Fig. 8-1. A minimum Z-80 system.

No WAIT conditions are necessary as the memory will always respond in time for data to be read, even at a 4-MHz clock rate. The output of the ROM is a three-state output, so that the lines are in a high-impedance state when the ROM is not being addressed. The eight output lines connect directly to the Z-80 data bus lines D0–D7. The output device is a quad latch whose four flip-flops are set by D0–D3 when an output operation is performed.

When the $\overline{\text{RESET}}$ switch is pressed, the $\overline{\text{RESET}}$ input goes low, initializes the CPU, and starts program execution at location 0 of the ROM. The ROM program is accessed by making memory requests MREQ and RDs, as no memory writes are possible, of course, with a read-only device. For this particular ROM, bringing both chip-select (CS) inputs to a logic 0 selects the ROM and gates the contents of the memory location addressed by A0–A8. The program is addressed by addresses $XXXXXXX000000000_2$ through $XXXXXXX-111111111_2$ where X may be any address, as address lines A9–A15 are not connected. (For clarity, all memory addresses would probably be in the range $0–511_{10}$.) The program probably requires some memory storage for variables, and this is provided by the 14

CPU registers. No memory stack is implemented, as no external RAM memory is provided and CPU registers cannot perform a stack function.

Data output is provided by the quad latch. Since this is the only I/O device in the system, *any* I/O instruction with *any* I/O address will address the latch and latch the contents of data bus lines D0–D3 when signals $\overline{\text{IORQ}}$ and M1 occur during an I/O cycle. Note that there is also no decoding of $\overline{\text{RD}}$ or $\overline{\text{WR}}$ and that even a *read* I/O instruction will output data to the latch. Output lines A, B, C, and D interface to the outside world.

The above example is admittedly a limited application of the Z-80, but it does serve to illustrate the simplest usable configuration of a Z-80 system. Even with this simple system, a program could be implemented to provide a variety of dedicated functions, such as:

1. Play music via the output latches
2. Provide simple digital-to-analog outputs (with a few additional external components)
3. Provide timing functions of almost any duration
4. Provide automatic telephone dialing (with additional external logic)

INTERFACING ROM AND RAM

A more usable system with ROM (or PROM) and RAM memory and limited I/O capability is shown in Fig. 8-2. A larger ROM (1K × 8) is used to provide 1024 bytes of program area. Two 256 × 4 bit high-speed RAMS (no WAITS necessary) are used to provide 256 bytes of read-write storage of *dynamic* variables. The RAM (and all system components) are three-state devices to enable "wire-ORing" all inputs and outputs to the data bus lines. One RAM reads and writes the four least significant bits of data from the data bus D3–D0, while the second RAM is used for D7–D4. A quad latch is used as before for I/O communication for 4-bit outputs from the CPU. In addition, four external input lines are sampled by gates G1 through G4.

The memory mapping for this configuration is shown in Fig. 8-3. The ROM memory area is located in locations 0000H through 3FFH. The RAM memory area is located at locations FF00H through FFFFH (256 locations). Address lines A10 through A14 are not used. Whenever address line A15 is a 0, ROM memory is being addressed, and whenever A15 = 1, RAM memory is being accessed. The I/O addresses in the Z-80 are separate from memory addresses (as opposed to a *memory-mapping* I/O). As in the previous exam-

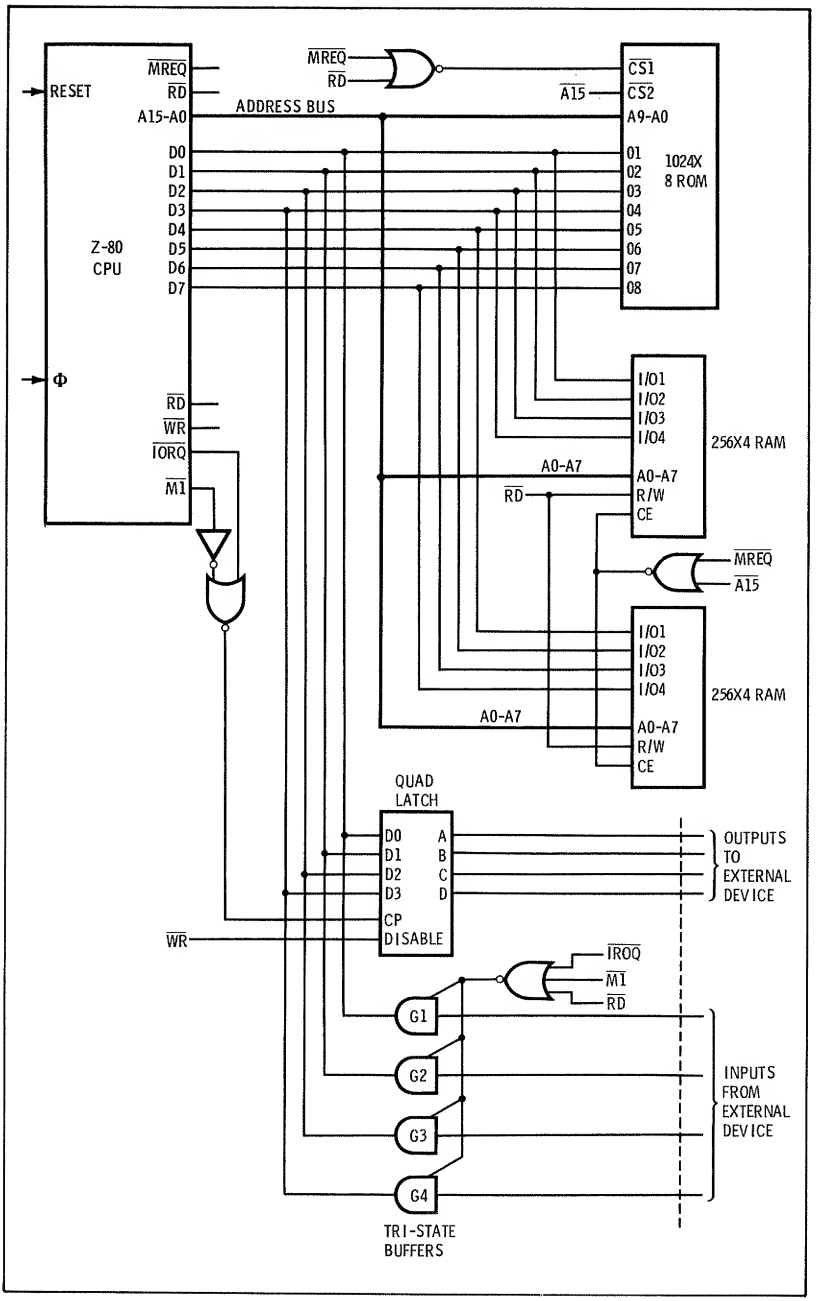


Fig. 8-2. Z-80 interfacing with RAM and ROM.

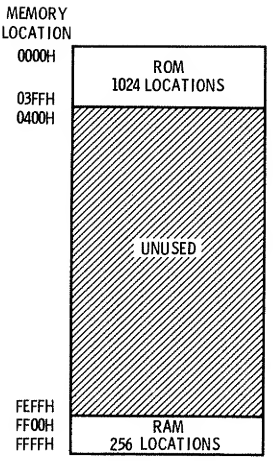


Fig. 8-3. RAM, ROM memory mapping.

ple, any I/O device address will address either the input or output I/O devices.

When the RESET switch is pressed, program execution starts at location 0 in ROM. The program can address RAM by addressing locations FF00H through FFFFH and can utilize a stack area by setting the stack pointer somewhere in this region. The RD signal to ROM is somewhat redundant in that all memory accesses to ROM *must be* reads. The RD/WR input to the RAMs is derived from the $\overline{WR}$ signal from the CPU. I/O inputs are handled a similar way as in the previous example. When an IN instruction is executed, $\overline{IORQ}$ and $\overline{M1}$ become active and the $\overline{WR}$ signal is also active. Data bus outputs D3–D0 are latched into the output latches during the output cycle. When an OUT instruction is executed, $\overline{RD}$ goes active, and enables the program to sample the input data lines I0–I3. The format of the output and input data is shown in Fig. 8-4. Data bits seven through four are ignored on output. For input, data bits D7–D4 will be zeros.

The system shown in Fig. 8-2 is an extremely powerful system even with the minimum memory configuration. Because it allows both the input and output, this system could be used to:

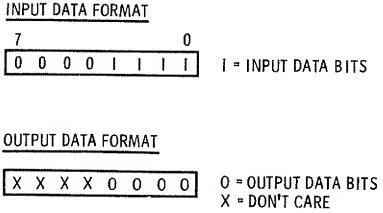


Fig. 8-4. Input- and output-data formats RAM/ROM configuration.

1. Decode bcd inputs and output bcd data in sequence after processing
2. Use the switch inputs for burglar or fire-alarm sensing with appropriate signal outputs
3. Input pulse-rate data representing instantaneous speed or other analogs and process the data
4. Provide digital-to-analog input decoding and analog-to-digital outputs

DYNAMIC MEMORY INTERFACING

Dynamic RAM memory is interfaced in much the same fashion as static RAM memory insofar as memory reads and writes for data and operands are concerned. Due to the electrical requirements of the RAM, however, every cell in the RAM must be refreshed periodically. Essentially, this means performing a read cycle for the cell without accessing the data from the cell about every 2 milliseconds. Through the R register in the Z-80, a means is provided to generate the refresh cycle automatically. At every $\overline{M1}$ time during an instruction cycle, signals $\overline{M1}$ and memory request $\overline{MREQ}$ become active to signal the external dynamic RAM memory that *one* refresh cycle may take place. The RAM then performs a refresh utilizing the current address on the data bus from the R register. Since the R register is continually sequencing from 0 to 255 in modulo 2^8 fashion every M1 cycle, a new refresh address is continually available to the dynamic RAM memory.

Fig. 8-5 shows a 4096-byte memory made up of eight 4096 by 1 dynamic RAMs. Each RAM has 12 address inputs split between six *row* inputs and six *column* inputs. The requirements for refresh are that within a 2-millisecond period each of the 64 possible rows are addressed. Since this cannot be assured by random access of the data, as in program execution, it must be systematically performed. To accomplish this, signals $\overline{RFSH}$ and $\overline{MREQ}$ are ANDed as shown. When both signals are false, signal $\overline{CE}$, chip-enable refresh, goes active and a read is performed for each of the eight chips, using address lines A5–A0 as the row address. As 64 refresh cycles must be performed to refresh all of the cells within a chip, the average time to perform a complete refresh is $64 \times N$, where N is the average instruction time for the Z-80. With $N = 2.5$ microseconds, it will take 160 microseconds to refresh all 8-K bytes. Signal CE is also enabled by the normal nonrefresh read or write cycle of the Z-80, when one of the bytes is accessed for instruction execution, data retrieval or storage.

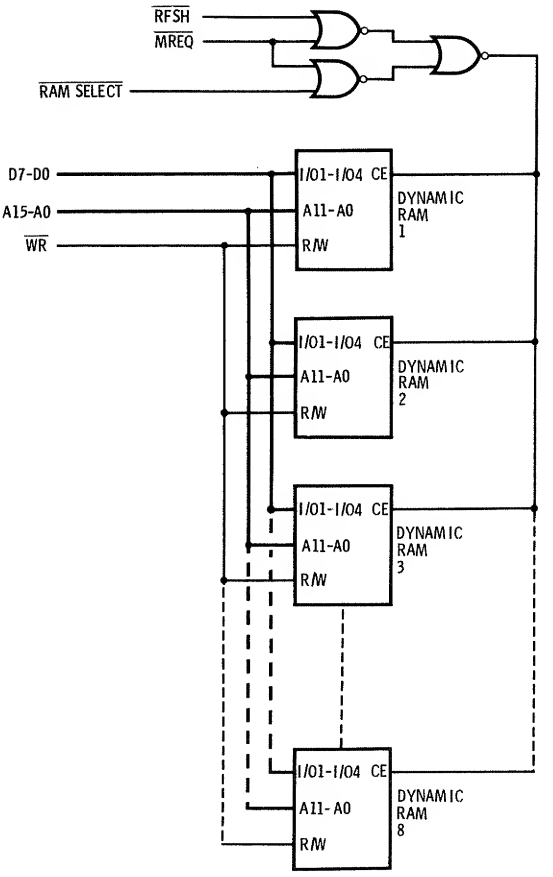


Fig. 8-5. Dynamic RAM refresh.

Z-80 PIO INTERFACING

The Z-80 PIO (Parallel I/O) is a 40-pin Z-80 compatible device that provides simple interfacing between the Z-80 and peripheral devices that accept 8-bit parallel data (see Fig. 8-6). Two 8-bit I/O ports are provided. They can be programmed for either input or output transfers. In addition to the two sets of eight bidirectional data lines (A7 – A0 and B7 – B0) there are two sets of two control lines used for *handshaking* between the I/O device and the PIO, A RDY and A STB, discussed later. Data is transferred between the PIO and the Z-80 CPU by data bus lines D7 – D0. Six control lines control PIO operations under program control from the Z-80 CPU. PORT B/A SEL selects port A or B. CONTROL/DATA SEL selects transfer of either control data or operand data to the PIO. Chip en-

able is the signal to the PIO indicating that the PIO address has been decoded in an I/O operation. $\overline{M1}$ is the CPU machine cycle one signal. $\overline{IORQ}$ and $\overline{RD}$ are the Z-80 signals related to any I/O operation. Three interrupt-control signals provide the interrupt INT, IEI, and IEO functions discussed in Chapter 7, that is, the eternal interrupt to the CPU and interrupt priority encoding. The clock input signal, Φ, is the clock signal from the Z-80 CPU.

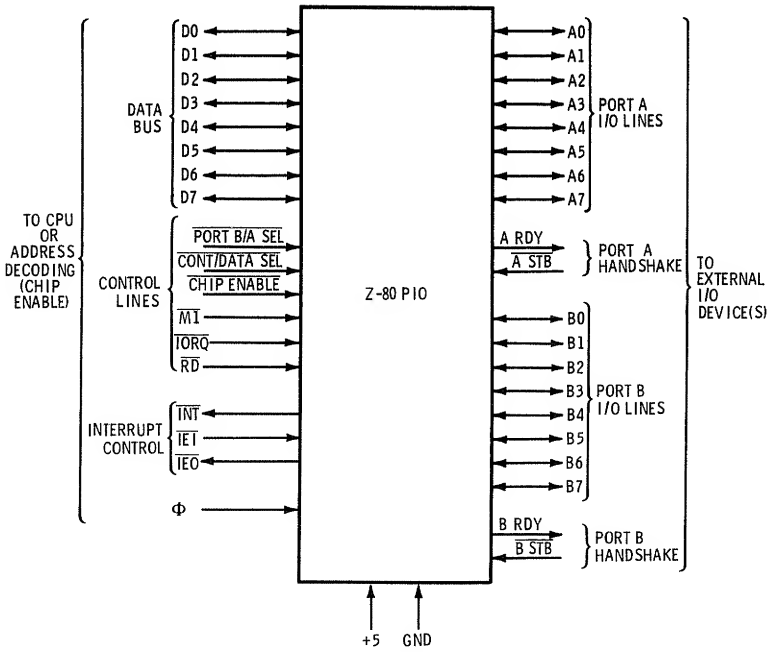


Fig. 8-6. PIO interface signals.

Internally, the PIO appears as shown in Fig. 8-7. Each port of the PIO has a number of registers associated with the port. The main controlling register is the 2-bit mode control register. It is set by addressing the PIO port and sending a control word from the CPU with the format shown in Fig. 8-8. The two most significant bits of the control word determine the mode as follows:

D7, D6	Mode
00	0 output
01	1 input
10	2 bidirectional
11	3 control

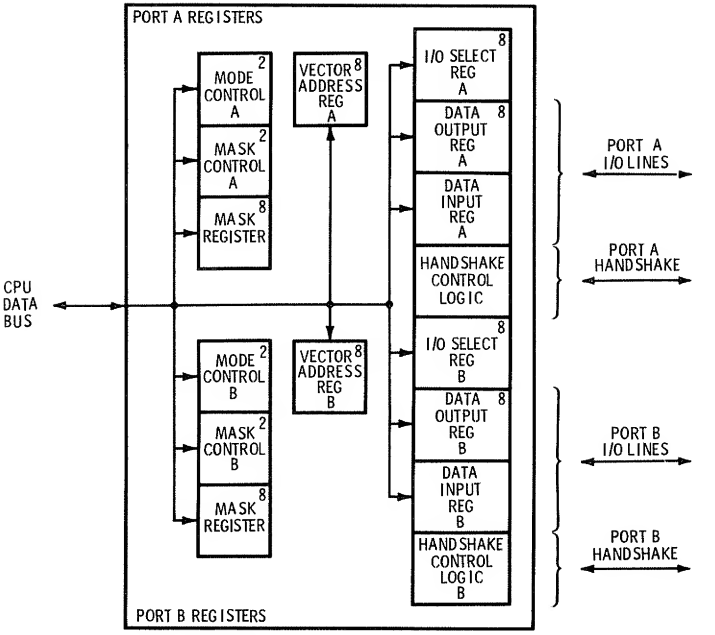


Fig. 8-7. PIO registers.

PIO MODE 0

Port A may be any mode, 0 through 3. Port B may be mode 0, 1, or 3. Mode 0 is the output mode of the PIO. In mode 0, the 8-bit data-output latch is active and the 8-bit data-input register is inactive. Data may be written to the data-output register by addressing the port and transferring eight bits of output data via an OUT instruction. Data may also be read back from the port by an IN instruction, although normally this would not be done as the program would always be cognizant of what data was written out. Data in the output register may be overwritten at any time by another OUT instruction.

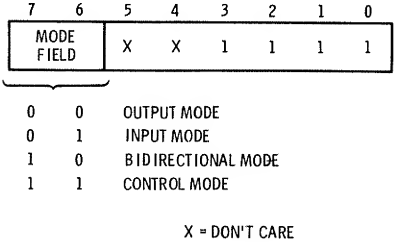


Fig. 8-8. PIO operation mode control word.

As data is written out to the PIO, the RDY signal associated with the port goes high, indicating to the external device that data is available on the port I/O lines. After the external device has read the data, it responds with signal $\overline{\text{STB}}$, resetting the port RDY signal and generating an interrupt (if the PIO has been programmed for an interrupt).

PIO MODE 1

PIO mode 1 is the input mode. If a port is in mode 1, the data-input register is active and the data-output register is inactive. The sequence of operations for inputting data into the PIO from an external device is as follows:

1. External device senses RDY line from PIO. If true, external device puts data on port I/O lines and momentarily brings down $\overline{\text{STB}}$ line.
2. Data is strobed into port data-input register. This resets the RDY line and causes an interrupt (if the PIO has been programmed for an interrupt).
3. Z-80 CPU reads the data from the PIO using an IN instruction with an I/O address of the PIO port.
4. RDY line is set by action of IN instruction, causing external device to ready next byte of data.

The actions in mode 1 are repeated for each byte of data to be read in. The input operation is initiated by an IN instruction in which the data is ignored as the RDY line is set for the first time.

PIO MODE 2

PIO mode 2 is the bidirectional data mode. Since mode 3 uses all four handshake lines, only port A may be used for this mode. The port A handshake lines are used for output operations and the port B handshake lines for input operations. When $\overline{\text{A STB}}$ is low, data from the data-output register of port A is gated onto the port I/O lines. When a $\overline{\text{A STB}}$ is high, data may be input into the data-input register by $\overline{\text{B STB}}$. Signals A RDY and B RDY may both be active at the same time, indicating that both output data is available from the PIO and that the PIO is ready to receive input data from the device.

PIO MODE 3

Mode 3 operations are set by addressing one of the two ports by an OUT instruction and transferring a second 8-bit control word

after the mode 3 control word has been transferred. In the second control word, each bit corresponds to a port I/O line as shown in Fig. 8-9. If a bit is one in the control, the corresponding port line is an input line. If a bit is a zero, the corresponding line will be an output line. The second control word sets the 8-bit Input/Output Select register shown in Fig. 8-7. Once mode 3 is set, data may be read or written to the port at any time. No handshaking signals are active; the STB signal is not used and the RDY signal is always low. Outputting data to the port will affect those lines programmed as outputs, while inputting data will read all lines, including those programmed as outputs.

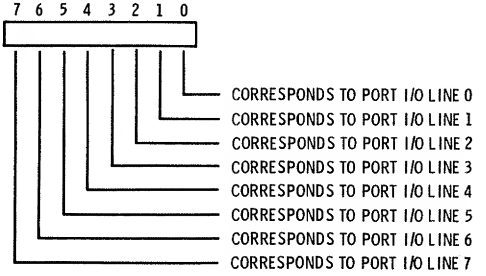


Fig. 8-9. PIO mode 3 input/output programming.

PIO INTERRUPTS

Each port of the PIO may be programmed to provide an external interrupt to the Z-80 CPU for input or output operations. When an OUT instruction with the port address is executed and the 8-bit control word shown in Fig. 8-10 is output to the PIO port, any subsequent mode 2 interrupts generated from the PIO port will use the interrupt vector of the control word which is stored in the port. Chapter 7 describes how the Interrupt Vector Table address is computed using the contents of the I register and the externally supplied vector. In this case, the PIO control word supplies the least significant 8 bits of that vector. Bit 0 is always a one, as is consistent with mode 2 interrupt operation. The PIO will *only* operate in the CPU mode 2 interrupt function and not in mode 0.

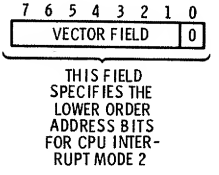


Fig. 8-10. PIO interrupt vector control word.

In conjunction with the port interrupt vector address register, each PIO port has an interrupt control word register of 2 bits and an interrupt mask register of 8 bits. The interrupt control word register holds data relating to the interrupt control word shown in Fig. 8-11. The interrupt control word is transferred to a port by addressing the port with an OUT instruction and transferring the control word. Bit seven of the control word controls the interrupt of the PIO. If bit seven is a one, the port will generate an interrupt; if reset, the port will not generate an interrupt. As previously discussed, the interrupt occurs on the rising edge of the STB signal for modes 0, 1, and 2. Bits six, five, and four are used only for PIO mode 3. Bit five defines the active state for the port I/O lines; if a 1, the active state is a high state. Bit six specifies either an AND or OR function for interrupt operation. If bit six is a 1, all bits must go to an active set (high or low) before an interrupt is generated. If bit six is a 0, any bit in the active state will generate an interrupt. The port lines that are monitored for the AND or OR condition are further defined for a mask. If bit four is a 1 after the interrupt control word has been received by the PIO, then the next word sent to the PIO must be a control word mask which is loaded into the port interrupt mask register. If a bit position is a 1 in the mask, then the corresponding line will be used as an active line for interrupt generation.

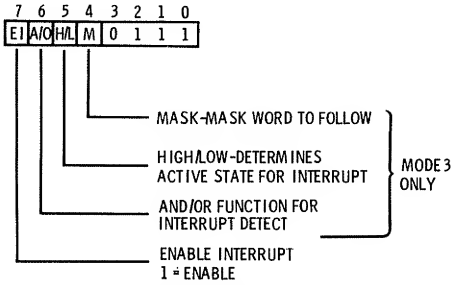


Fig. 8-11. PIO interrupt control word.

PIO INITIAL CONDITIONS

The PIO is initialized on a power-up or $\overline{M1}$ condition without $\overline{RD}$ or $\overline{IORQ}$. The latter condition enables a reset without power-down and without adding additional signals to the PIO for reset. The initial PIO conditions are as follows:

1. Port interrupt enable flip-flops, output registers, and mask registers reset
2. Mode 1 selected

3. Port I/O lines set to high-impedance state
4. Handshake signals low (inactive)
5. Vector address registers not reset

Z-80 PIO CONFIGURATION

Fig. 8-12 shows one PIO replacing the four data-output lines and four gated input lines of Fig. 8-2, effectively doubling the I/O capability of the Z-80 system and providing complete interrupt control. Data lines from the Z-80 bus are input to the PIO data line inputs. Port A output lines are used to write to an external device while port B input lines are used to read to an external device. The two sets of handshake lines are used in the same fashion. The INT line from the Z-80 PIO is input to the CPU directly. Since there is only one PIO in the system, there is no daisy-chained interrupt priority and IEI is set to VCC. IEO is not used. Inputs $\overline{M1}$, $\overline{IORQ}$, and $\overline{RD}$ are connected directly to the equivalent Z-80 signals.

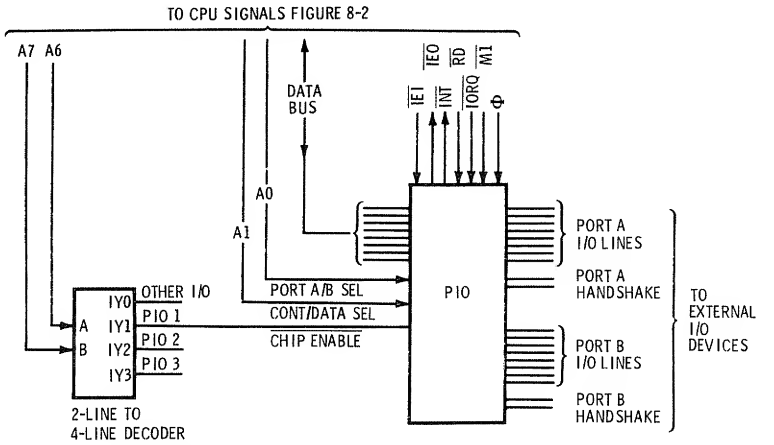


Fig. 8-12. PIO use in minimum configuration system.

As the PIO is the only I/O device in this system, there is no need to decode the I/O address. However, a decode is shown for larger systems. The PIO address is determined by the three most-significant bits of the I/O address A7 and A6. This scheme would allow for four PIOs or, as shown, three PIOs and other I/O addresses in the range 00000000 through $00XXXXXX_2$. The output of the two to four demultiplexor enables the PIO for I/O address $01XXXXXX_2$. The port A/B select line is connected to A0 and the control/data select line to A1. The address mapping for the addressing configuration is shown below:

IN/OUT IO Address	Meaning
00000000_2	Non-PIO addresses
00111111_2	
01XXXX00	Port A, data
01XXXX01	Port B, data
01XXXX10	Port A, command
01XXXX11	Port B, command
10000000	Other PIO expandability
↓	↓
11111111	

To output and input data to the I/O devices not under interrupt control, the following steps must be taken:

1. Reset the PIO (power on). This clears the interrupt enable and PIO interrupt vector registers.
2. Load interrupt control word or 07H into CPU register R. Output to devices 01000010_2 and 01000011_2 with OUT instruction. This disables PIO interrupts in both ports.
3. Load operating mode control word 00001111_2 into CPU register R. Output to device 01000010_2 with OUT instruction. This sets up the A port as an output port.
4. Load operating mode control word 01001111_2 into CPU register R. Output to device 01000011_2 with OUT instruction. This sets up the B port as an input port.
5. Input data from device 01000001_2. This inputs data from port B. Initial data is discarded, but the B RDY line is activated, informing the external device that the CPU is ready for data.
6. Port A is now ready to output data and port B is ready to input data. Since no interrupts are programmed, output must be timed so that the external device has sufficient time to respond to the output and to provide input data. A timing loop must be included in both the *read* and *write* I/O drivers for this PIO.

To perform I/O under interrupt control, the interrupt vector registers must first be output to the PIO and the proper interrupt control words must be output. If the interrupt processing routine address for the output device was at FF00H and the interrupt processing routine address for the input device was at FF02H, then the CPU I register must have been loaded with FFH before I/O activity. Next, sometime before the first I/O activity the PIO interrupt vector registers must have been loaded as follows:

—Load interrupt vector control word 00H into CPU register R.

Output to device 01000010_2 with OUT instruction. This sets the port B interrupt vector register to 02H.

Finally, the interrupt control word of 80H must have been output to device addresses 01000010_2 and 01000011_2. This would enable interrupts for both port A and port B. Interrupts would occur on port A each time the external device strobed into the output data (A STB went momentarily low) and on port B each time the external device strobed data into the PIO input register (B STB went momentarily low). Interrupts for port A would vector to the address specified in FF00H and interrupts for port B would vector to the address specified in FF02H as described in Chapter 7.

The above description illustrates the interfacing for one Z-80 PIO. The configuration shown could be used for a variety of uses including Teletype I/O, keyboard decoding, high-resolution a-to-d or d-to-a I/O, and 16-line process-control applications. Using similar procedures to those shown above, the reader can see how multi-PIOs and additional ROM, PROM, or RAM memory can easily be added to the system. Further examples of microcomputer systems built around the Z-80 family of components will be provided in Section III.

SECTION II

Z-80 Software

CHAPTER 9

Z-80 Assembler

The previous section described the hardware aspects of the Z-80, including the inherent instruction set of the microprocessor. Section II describes how to use that instruction set efficiently to build sets of instructions to perform software functions such as multiplication, division, double and multiple-precision arithmetic, and table and string manipulation. To facilitate the writing of software programs, an *assembler* program is employed. The assembler provides an easy way to automatically *assemble* machine language instructions from a higher-level *symbolic assembly language.*

MACHINE LANGUAGE

Machine language is the most rudimentary form of any program. It consists of the actual machine language operation codes and operands necessary to implement the instructions of the program, expressed in binary or hexadecimal numbers. Suppose, for example, that a short program is required to add the numbers from one to ten. An extremely inefficient way to perform this task is shown below.

```
XOR A
ADD A,1
ADD A,2
ADD A,3
ADD A,4
ADD A,5
ADD A,6
ADD A,7
ADD A,8
ADD A,9
ADD A,10
```

The program consists of an instruction to clear the A register (the XOR) and a succession of ten immediate instructions to add the numbers one through ten to the contents of the accumulator. The program is written in the *mnemonics* that Zilog uses for the equivalent machine-language code, along with the register to be used and the immediate 8-bit data value. To assemble the equivalent machine-language code, one would have to look up the hexadecimal form of the operation code and the format of the instruction and write it beside each mnemonic representation of the instruction as shown in Fig. 9-1. The figure shows that the XOR A is a one-byte instruction of the form $10101RRR_2$, where R is the register required. In this case, $R = 111_2$, indicating A. The ADD format is a two-byte instruction of the form 11000110_2, followed by an 8-bit field representing the 8-bit immediate operand. The equivalent machine-language instruction for an ADD A,8, for example, is the op code 11000110_2 or C6H, followed by 00001000_2 or 08H.

The entire program representing the addition of one through ten could be loaded into the Z-80 microcomputer by means of a control panel (if the microcomputer has one) or *monitor* program and then executed. The actual numbers that would be keyed in are the numbers shown in the left-hand column of Fig. 9-1, twenty-one 8-bit bytes of machine code representing the program.

Let us take another program example to illustrate the machine-language assembly process once again. This time we will assemble

MACHINE CODE	INSTRUCTION FORMAT	INSTRUCTION
AFH	1 0 1 0 1 | 1 1 1	XOR A
C6	C6H	ADD A,1
01	01H	
C6	C6H	ADD A,2
02	02H	
C6	C6H	ADD A,3
03	03H	
C6	C6H	ADD A,4
04	04H	
C6	C6H	ADD A,5
05	05H	
C6	C6H	ADD A,6
06	06H	
C6	C6H	ADD A,7
07	07H	
C6	C6H	ADD A,8
08	08H	
C6	C6H	ADD A,9
09	09H	
C6	C6H	ADD A,10
0A	0AH	

Fig. 9-1. Manual assembly process program 1.

a program to add the numbers from one to ten in a slightly different implementation. We will use the A register to hold the total as before, but we will let the B register hold the current number to be added. This will vary from 10 to 1 as we go in reverse, adding 10, then 9, then 8, and so forth down to 1. At that point, we will detect that the next number to be added is 0 and stop. The program in Zilog mnemonics looks like this:

```
        XOR     A           CLEAR A
        LD      B,10        SET COUNT TO 10
LOOP    ADD     A,B         ADD NEXT NUMBER
        DEC     B           PREPARE NEXT NUMBER
        JP      NZ,LOOP     JUMP IF NUMBER≠0
        HALT                HALT
```

The A register is first cleared by the XOR A instruction. Next, the B register is loaded with 10 by the LD B,10 instruction. The next three instructions comprise a *loop*. As long as B holds a number from 10 to 1, the contents of B will be added to A (ADD A,B), the contents of B will then be decremented by one (DEC B), and the jump will be made to the first instruction of the loop which is *labeled* "LOOP" as a point of reference of where to return. When the B register is decremented, the Z flag is set if the result is zero and reset if the result is nonzero. If the B register is nonzero (9 through 1) the JP NZ,LOOP instruction will detect the nonzero (NZ) and jump back to LOOP. If the B register holds a 0, the Z flag is set and the conditional jump back to LOOP will *not* be made, causing the CPU to execute the next instruction (HALT).

Manually assembling the machine code for this program is a little more complicated than the preceding example. First of all, while the previous program could be *relocated* or *loaded* anywhere in memory, since the instructions contained no addresses, the second program *does* contain addresses (JP NZ, LOOP must specify the address of LOOP in bytes two and three of the instruction). A decision must therefore be made where in memory this program is to execute. We will arbitrarily choose location 0100H as the start. The next step in the assembly process is to calculate the length in bytes of each instruction and write it opposite each mnemonic. After this step, the program appears as shown below:

Location	Length		Instruction
0100H	1		XOR A
	2		LD B,10
	1	LOOP	ADD A,B
	1		DEC B
	3		JP NZ,LOOP
	1		HALT

Now the locations of each instruction can be filled in, using the length to adjust each location. The location *always* specifies the *first* byte of the instruction.

Location	Length		Instruction
0100H	1		XOR A
0101H	2		LD B,10
0103H	1	LOOP	ADD A,B
0104H	1		DEC B
0105H	3		JP NZ,LOOP
0108H	1		HALT
0109H			

As a double check on the accuracy of this step, the total length of the program from the location column (0109H–0100H = 9 bytes) can be compared with the total number of bytes from the length column, nine. Now the instruction formats can be filled in, as shown in Fig. 9-2. The only difficult instruction is the JP NZ,LOOP. This is a three-byte instruction with the last two bytes specifying the conditional jump address. Since the jump is to LOOP, which is at location 0103H, this address must go into bytes two and three in reverse order 03H, 01H, as is the format from time immemorial (or at least since the 8008).

LOCATION	LENGTH	MACHINE CODE	INSTRUCTION FORMAT		INSTRUCTION
0100H	1	AF	10101 111		XOR A
0101H	2	060A	00 000 110 0000 1010		LD B,10
0103H	1	80	10000 000	LOOP	ADD A,B
0104H	1	05	00 000 101		DEC B
0105H	3	C20301	11 000 010 0000 0011 0000 0001		JP NZ,LOOP
0108H	1	76	0111 0110		HALT
0109H					

Fig. 9-2. Manual assembly process program 2.

Although it is feasible to assemble long programs by manual methods, it is extremely uneconomical. There is too much of a chance for error in calculating locations, filling in instruction fields, and formatting addresses. In addition to the certainty of rote errors, there are several other factors that make machine-language operations unworkable. The most important of these is relocatability. Program two could execute only at location 0100H. To execute at another location, the address in the JP instruction would have to be changed. In larger programs, many addresses would have to be refigured and manually assembled. A second factor is ease of editing. Few programs run the first time and most require several iterations before the program performs the way that was expected. Each iteration involves adding, deleting, or modifying instructions of the pro-

gram, necessitating recalculating addresses where they are used in the program.

THE ASSEMBLY PROCESS

Because of the inherent limitations of manual assembly, all microcomputer manufacturers offer an *assembler program* to automatically perform the machine-language function from symbolic assembly language. Many times the assembler may be run on the microcomputer itself. In this case, the assembler is a *resident* assembler. In a few cases, the assembler must be run on another computer, typically an IBM 360/370 configuration. In the latter case, the assembler is a *cross-assembler*. In either case, the assembler quickly assembles programs written in Z-80 or other *source* assembly languages, producing an *object module* representing the machine code, and a *listing* of the program in both assembly and machine language form. A few of the features that an assembler provides are:

1. Symbolic representation of locations, operation codes, and arguments
2. The ability to intermix comments with the symbolic form of the instruction
3. Automatic assembly of forward and backward references to symbolic locations
4. Automatic representation of various number bases
5. Expression evaluation
6. Pseudo-operations or nongenerative assembler instructions that define locations, equate symbols, reserve memory, and other convenient features

ASSEMBLY FORMAT

The mnemonic representation of instructions have been used throughout this text. They are simply a convenient way to write down the instruction as it is much simpler to write "ADD A,B" than to write "add the contents of the B register to the contents of the A register." The mnemonics used for the Z-80 in this text closely follow the ones used by Zilog. There are some slight differences in representation of addressing types. The tables in Chapter 5 or Appendix C list all instruction mnemonics and the possible addressing formats. Other microcomputer manufacturers described in this book may use somewhat different mnemonics in their documentation for their products.

The standard assembly-language format used in this book is shown in Fig. 9-3. There are four columns, the label column, the

op code column, the arguments column, and the comments column. Each representation of a Z-80 instruction must have an operand. Most instructions have arguments, such as "LD (HL),R" where "(HL),R" are the two arguments. Instructions such as EI or HALT have no arguments. The label field is optional. When a label is present, it may be one to six alphabetic or numeric characters, the first character of which must be alphabetic. The optional comments

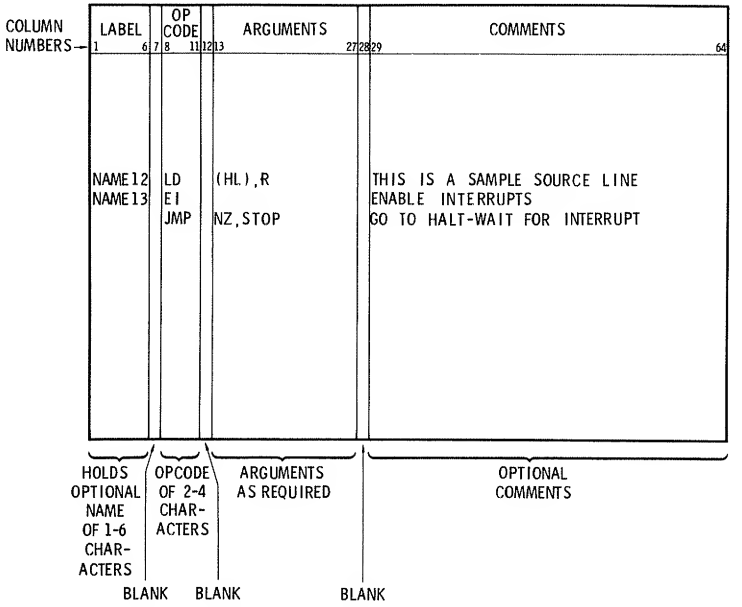


Fig. 9-3. Typical Z-80 assembly language format.

column describes the action of the instruction as was shown in Fig. 9-2. The four columns make up an assembly language *line*. In general, the length of assembly language lines has been determined by the length of lines on the input devices such as teletypewriters and punched-card readers. In actual practice, as in the assemblers discussed in the manufacturers section of this book, Section III, the end of the line is represented by a carriage return, line-feed code, or similar device-oriented condition.

In general, each assembly language line (or *source* line) represents the complete set of information about one Z-80 instruction. Each line will generate from one to four bytes representing a Z-80 instruction. One of the several exceptions to this rule is a comment line, which is originated with a semicolon and is *nongenerative;* it generates no machine-language code but serves for reference only.

A partial typical page from an assembler listing is shown in Fig. 9-4. The information printed to the left represents the source line number, the memory location for the first byte of the instruction, and up to eight hexadecimal digits for the machine-language code of the instruction (two hexadecimal digits represent one byte). This is information generated and listed by the assembler. The information printed to the right represents a direct *image* of the source line itself. Additional data printed on the listing would consist of *diagnostic*

```
105                 ;*********************************************************
106                 ;*  BXASH-00-00                                          *
107                 ;*                                                       *
108                 ;*  FUNCTION: THIS SUBROUTINE CONVERTS AN 8-BIT BINARY VALUE*
109                 ;*       IN THE C REGISTER TO TWO ASCII HEXADECIMAL DIGITS. *
110                 ;*                                                       *
111                 ;*  CALLING SEQUENCE:  (HL)=BUFFER AREA POINTER          *
112                 ;*                     (C)=8-BIT VALUE TO BE CONVERTED   *
113                 ;*                     CALL BXASH                        *
114                 ;*                     (RTN W/CHARACTERS IN BUFFER, BUFFER +1*
115                 ;*                         AND HL INCREMENTED BY 2)      *
116                 ;*********************************************************
117                 ;
118  1036  3EF0     BXASH   LD    A,F0H
119  1038  A1               AND   C             MASK 1
120  1039  0F               RRCA
121  103A  0F               RRCA
122  103B  0F               RRCA
123  103C  0F               RRCA                ALIGN FOR CONVERSION
124  103D  CD4710           CALL  CVERT         CONVERT
125  1040  3E0F             LD    A,FH          MASK 2
126  1042  A1               AND   C             GET SECOND CHARACTER
127  1043  CD4710           CALL  CVERT         CONVERT
128  1046  C9               RET
129                 ;
130  1047  C630     CVERT   ADD   A,30H          CONVERT TO 0-15
131  1049  FE10             CP    10            TEST FOR 0-9
```

Fig. 9-4. Typical Z-80 listing.

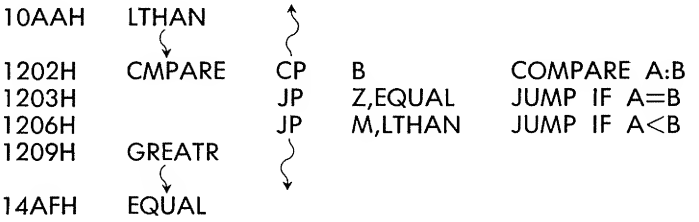
messages indicating assembly errors such as a reference to an undefined, or multiply-defined symbol, invalid arguments such as invalid hexadecimal digits and the like. Since the listing format is dependent on the microcomputer system and the kind of assembler, this discussion is meant to provide a general picture of how a typical listing would appear and is not meant as a detailed guide.

SYMBOLIC REPRESENTATION

The label column of the source line represents the name of the location. A program could be written with references only to *absolute* locations such as "LD A,123AH." In this case, however, it would be necessary to know the exact location to be used, necessitating definitions of numeric addresses to be used for variables and con-

stants. It is much more convenient to write "LD A,RESULT" than to specify an absolute location. The assembler will automatically resolve the symbol "RESULT" into the equivalent machine-code address. The references to *symbolic* rather than absolute locations may be either *backward references* to previously defined symbols, or *forward references* to yet-to-be-defined symbols. Let's see how the assembler resolves the symbols with a short program.

The program below compares the contents of the A register to the contents of the B register by a compare instruction. It then branches (jumps) out to three addresses dependent on whether A<B, A=B, or A>B, represented by location LTHAN, EQUAL, or GREATR. LTHAN is a backward reference, while EQUAL and GREATR are forward references.



The arrows represent instructions not defined. The locations to the left represent the locations after assembly. Most assemblers make two *passes*. The first decodes the mnemonics, constructs as much of the instruction as possible, counts the bytes in the instruction, and constructs a *symbol table* representing all labels and symbols in the program. The second pass resolves all addresses by the symbol table. The reason for *two* passes is that forward references cannot be resolved until the symbol is encountered. After the first pass for the above program, the symbol table will show:

Symbol	Value
CMPARE	1202
EQUAL	14AF
GREATR	1209
LTHAN	10AA

On the second pass, the values of EQUAL and LTHAN will be filled into the JP instructions at 1203H and 1206H.

Certain symbols in the Z-80 system are *reserved* and cannot be used by the programmer. The assembler has set these symbols aside to define registers or addressing modes. Many of these symbols appear in the instruction formats of Appendix C. Reserved words in the Z-80 system would include:

Register Names:	A, B, C, D, E, F, H, L
Register Pair Names:	AF, BC, DE, HL, IX, IY, SP, AF′
Condition Code Flags:	C, NC, Z, NZ, M, P, PE, PO

REPRESENTATION OF NUMBER BASES

Another assembler feature present on all assemblers is the ability to convert from one number base to the other. This means that arguments for instructions may be specified in the most convenient base. The ADD A,N instruction, for example, adds an 8-bit immediate value to the contents of the A register. Binary, decimal, or hexadecimal values of N may be specified by a suffix of B, no suffix, or H to enable specifying any of the three number bases: "ADD A,100", "ADD A,64H", and "ADD A,01100100B" all amount to the same thing, adding 100_{10} to the contents of the A register. These three suffixes will be used in the examples of Section II, although the formats actually used in a particular Z-80 assembler undoubtedly will be different.

EXPRESSION EVALUATION

Most assemblers have limited expression capability. Expressions may consist of symbolic and literal data and in more sophisticated assemblers, *absolute* and *relocatable* symbols. Expression operators allow addition, subtraction, multiplication, and in some cases, division and shifting. The operators are usually represented by predictable symbols, such as "+", "−", "*", and "/" for addition, subtraction, multiplication, and division. Elaborate expressions find little use in assembly language programs and in some cases may overpower the assembler, but simpler expressions may be used to assemble the length of a table, calculate system parameters, and create fields within data words. Examples will be given in this and other chapters.

PSEUDO-OPERATIONS

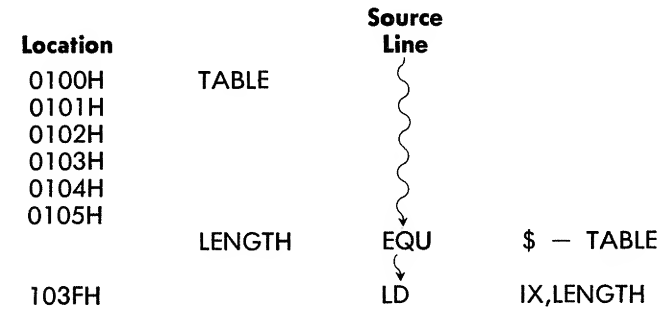
In each source line, the portion responsible for generation of the instruction operation is the op code. There are some assembler operation mnemonics, however, that do not generate machine-language instructions but, rather, inform the assembler of special actions to be taken. These operation mnemonics are called *pseudo-ops,* since they are not truly operation codes that represent valid machine-language instructions. The pseudo-ops discussed here are similar to those in all assemblers. As they are shown, the parentheses represent an optional label.

Label (if any)	Pseudo-Op	Argument (if any)
	ORG	N
	END	
NAME1	EQU	NAME2
(NAME1)	DEFB	N
(NAME1)	DEFW	N
(NAME1)	DEFS	N
(NAME1)	TXT	STRING

The ORG pseudo-op establishes the origin of the program. When, for example, "ORG 1200H" is used before the first source line of code, the assembler *location counter* will be set to 1200H. Subsequent instructions will advance the location counter by the number of bytes in each instruction so that the assembler may keep track of symbol locations and the current instruction location. The ORG may also be used within a program at any time to start assembly from a new location.

The END pseudo-op is the last statement in a program and signals the assembler to start pass two or to end the assembly process.

The EQU pseudo-op equates a label to another label or a numeric value. The EQU is used for convenience in assigning recognizable names to constants or expressions. An example of an EQU representing the length of a table is defined below. Here "$" represents the *current assembler location* (the contents of the assembler location counter).



The length of the table in this example will be 0106H (the current location counter)—0100H (the start of the table) or 6 bytes. The EQU *does not generate code*, but makes an entry in the symbol table under "LENGTH" for a value of 6. Later in the program, when the 16-bit immediate instruction LD IX,LENGTH is encountered, the assembler searches the symbol table for the symbol LENGTH and resolves it with the value 6. Execution of LD IX, LENGTH will then load 6 into the IX register.

The pseudo-ops DEFB and DEFW define constants and variables in the program. The argument for the DEFB is a numeric or symbolic expression that can be resolved in eight bits. The argument for DEFW must be resolved in sixteen bits. Both pseudo-ops are necessary because without them the assembler could not generate tables of data, constants, or locations for variables. The following source lines generate a table of ten bytes, each byte representing data from 1 to 10.

```
0100    01      TABLE   DEFB    1
0101    02              DEFB    2
0102    03              DEFB    3
0103    0405            DEFW    0405H
0105    06              DEFB    6
0106    0708            DEFW    0708H
0108    09              DEFB    1001B
0109    0A              DEFB    AH
010A
```

DEFS is a pseudo-op that *reserves* a number of bytes. In many cases, it is necessary to set aside a block of memory without actually filling it with meaningful data, as in allocation of I/O buffers and working storage areas. The effect of DEFS is to increment the assembler location counter by the argument, which represents the number of bytes to be reserved. When the assembled *object module* is loaded by the loader program after assembly, the block of storage allocated by the DEFS will not be affected and will retain the meaningless data in the memory area before the load. An alternative way to reserve storage is to use an ORG pseudo-op. Both of the statements below reserve 22H bytes starting at location 1234H.

```
1234H   BUFFER  EQU     $
1234H           DEFS    22H
1256H   NEXTI   LDD
                 ↓
-------------------------------------
1234H   BUFFER  EQU     $
1256H           ORG     $ + 22H
1256H   NEXTI   LDD
                 ↓
```

The last pseudo-op discussed here, TXT, is similar to the DEFB and DEFW in that it generates data for use by the program. The data in this case is ASCII text data. ASCII representation is used for most I/O devices and is shown in Appendix E. Alphabetic, numeric, and special characters must be encoded in ASCII format before being transferred to the I/O device for printing, display, or punching. The TXT pseudo-op generates one ASCII character for each text character in the argument string. The argument string is

started by any character and is encountered by the same character. It is convenient to use unusual characters as the *delimiters*.

```
0100    43555253    TXT    $CURSE YOU RED BARON$
0104    4520594F
010B    55205245
010C    44204241
0120    524F4E00
0124
```

Delimiter

The pseudo-ops above are some of the most commonly seen and will be used in the examples of Section II. The actual pseudo-ops used in Z-80 microcomputers software will vary, however, and the reader must refer to the manufacturer's literature for the mnemonics and formats used.

ASSEMBLY MECHANICS

Once a program has been written, the actual assembly mechanics are quite easy. The source statements are entered via the keyboard and a copy of the source lines is recorded on some type of I/O medium such as paper tape, magnetic tape, or floppy disc. In many cases, a utility program called an *editor* is used to transfer the keyboard input to the storage medium. After the program has been copied onto the medium, the assembler is loaded into the microcomputer if a resident assembler is being used, or into the host computer if a cross-assembler is employed. The assembler will then read the source from the storage medium for the first pass. If paper- or magnetic-tape cassettes are used as the storage medium, the paper tape or cassette may then have to be repositioned manually to the start of the source image; in other cases, the system will automatically restart from the beginning of the input medium. The assembler then executes the second pass producing a listing such as the one shown in Fig. 9-3 and an *object module*. The object module is essentially the machine-language code in a special *loader format*. The object module may physically be paper tape, magnetic tape, or floppy disc. The object module output of the assembler can now be loaded into microcomputer memory by the loader utility program and, after the load, be available for execution.

As previously mentioned, few programs run the very first time, and subsequent reassemblies, loads, and executions will undoubtedly have to be performed until a final version is produced. For each iteration (and in some systems there are dozens), the assembler greatly simplifies the coding process.

CHAPTER 10

Moving Data—Load, Block Transfer, and Exchange Groups

This chapter discusses one of the most basic operations in any computer system, moving data between CPU registers and external memory, or between two areas of external memory. The moves may be eight or sixteen bits at a time. The moves may involve transferring data from one location to another, copying the contents of a source location to a destination location, or they may involve exchanging the contents of both locations. Some of the moves involve storage and retrieval of data from the portion of external memory used as a stack. The most sophisticated of the moves transfers up to 64-K bytes in one instruction.

8-BIT MOVES

The 8-bit load group allows data to be moved from a CPU register to memory or from memory to a CPU register in a variety of addressing modes. Moving data to, or from, the A register is a special subset in this group. The A register is given precedence because it is the primary register used for arithmetic, logical, and shifting operations in the 8080 and 8008; and these uses still carry over to the Z-80.

Any of the general-purpose CPU registers can be loaded with the contents of another CPU register or immediate value by a LD R,R′ or LD R,N instruction, respectively. The following code loads A, B, C, D, and E with 0 through 4, respectively, and then reverses the order (4 through 0) by LD R,R′ instructions.

```
LD A,0          LOAD 0
LD B,1          LOAD 1
LD C,2          LOAD 2
LD D,3          LOAD 3
LD E,4          LOAD 4
LD H,A          SAVE A
LD L,B          SAVE B
LD A,E          E TO A
LD B,D          D TO B
LD D,L          B TO D
LD E,H          A TO E
```

When data is to be moved from memory to CPU registers, there are several methods that can be used to implement the move. These methods are "mirrored" for moving the data from CPU registers back to memory, so that a good way to illustrate the move is to show how data can be moved from one block of memory to another. Obviously, the easiest way to implement a move of this kind is with the block-transfer instructions, but the discussion of this group will be left until later in the chapter. The general methods for moving eight bits of data at a time from memory to CPU registers or back again are:

1. Using any CPU registers and HL as a pointer in register indirect mode
2. Using indexed addressing with any CPU registers
3. Using direct (extended type) addressing *with the A register only*
4. Using BC or DE register indirect addressing *with the A register only*

We will discuss each of these methods in turn and illustrate 8-bit data movement to and from CPU registers with a short program for each method.

8-BIT MOVES USING HL

The following program loads the A, B, C, and D registers with four variables from memory labeled VAR1, VAR2, VAR3, and VAR4. Register pair HL is first set up as a pointer by a 16-bit load instruction that loads the start of the 4-byte block into HL. Each time the next variable is loaded, the HL register is incremented by one to point to the next byte of the block.

```
LD HL,START     POINT TO START
LD  A,(HL)      LOAD VAR1
INC HL          POINT TO START + 1
LD  B,(HL)      LOAD VAR2
```

```
        INC HL          POINT TO START + 2
        LD  C,(HL)      LOAD VAR3
        INC HL          POINT TO START + 3
        LD  D,(HL)      LOAD VAR4
        ↓
START   EQU $           THIS EQUATES START TO VAR1
VAR1    DEFS 1          THESE VARIABLES FILLED
VAR2    DEFS 1            IN WITH VALUES SOME-
VAR3    DEFS 1            TIME DURING PROGRAM
VAR4    DEFS 1            EXECUTION
```

Note that the above method worked quite well as the four variables were in one *contiguous* block. If the variables were in random locations, a little more work is involved as shown next in a short program that stores the contents of A, B, C, and D in four locations labeled STOR1, STOR2, STOR3, and STOR4. Each time a new register is stored, the HL register pair must be loaded with a new address since it cannot simply be incremented or decremented. Although there are many other ways to implement this problem in the Z-80, programs written for the 8008 had to use this method to access random data, as only the HL register pair was available as a pointer.

```
        LD HL,STOR1     STOR1 ADDRESS
        LD (HL),A       STORE A
        LD HL,STOR2     STOR2 ADDRESS
        LD (HL),B       STORE B
        LD HL, STOR3    STOR3 ADDRESS
        LD (HL),C       STORE C
        LD HL,STOR4     STOR4 ADDRESS
        LD (HL),D       STORE D
        ↓
STOR1   DEFB 0          THESE VARIABLES INITIALLY SET
          ↓               TO 0 BY DEFB. THEY WILL
STOR2   DEFB 0            BE FILLED WITH A–D.
        ↓
STOR3   DEFB 0
        ↓
STOR4   DEFB 0
```

8-BIT MOVES USING INDEX REGISTERS

The index registers IX and IY in the Z-80 are registers that are analogous to the HL register. Each of the index registers is a data pointer, but with an important difference. The effective address is obtained by adding an 8-bit displacement value to the contents of the index register. This means that within each instruction a dis-

placement can be added to the pointer to access data within a "block" of 256 bytes, starting from the location the index register points to −128 bytes and ending with the index register +127 bytes as shown in Fig. 10-1.

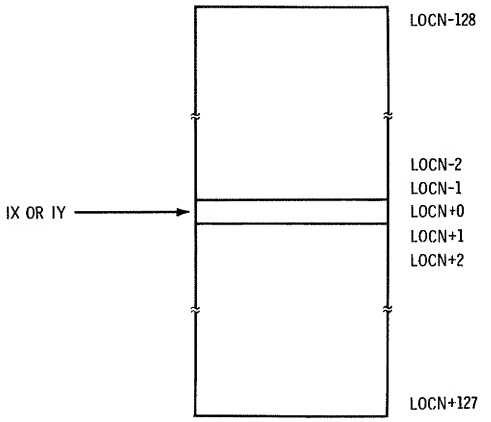


Fig. 10-1. Index register block access.

Suppose that the requirement was to store the A, B, C, and D registers into locations BLOCK − 4, BLOCK, BLOCK + 4, and BLOCK + 8, respectively. The following instructions would accomplish the task:

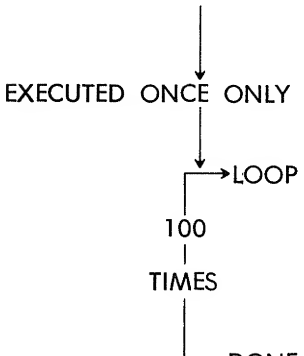
```
LD IX,BLOCK       POINT TO BLOCK
LD (IX - 4),A     STORE A INTO BLOCK - 4
LD (IX + 0),B     STORE B INTO BLOCK
LD (IX + 4),C     STORE C INTO BLOCK + 4
LD (IX + 8),D     STORE D INTO BLOCK + 8
```

The displacements in the third byte of the instruction would be −4, 0, 4, and 8, respectively. Here the process of storing data within the 256-byte block was made much more efficient than the example using the HL register pair pointer. Or was it? Let's compare the relative sizes and timing of the two programs. The first program using the HL registers used four three-byte instructions (LD HL,STORX) and four 1-byte instructions (LD (HL),D) for a total of sixteen bytes and 17 microseconds. The program above used five 3-byte instructions for a total fifteen bytes and 22.5 microseconds! It appears that the first implementation was faster and only slightly more expensive in terms of memory usage than the second. This is only one example of how execution speeds and memory storage requirements must be compared between one method of implementation and another if one is concerned about program efficiency.

If the IY register was to be used instead of the IX, the instruction format would be virtually identical, with "IY" substituted for "IX." The index register-oriented instructions can be used to advantage for moving data as in the following routine that moves the three bytes in BLK2 through BLK2 + 2 to BLK1 through BLK1 + 2. The move is implemented in reverse fashion starting at BLK2 + 2 and BLK1 + 2. IX holds the source pointer while IY holds the destination pointer. Both index registers are decremented by a DEC IX or DEC IY instruction.

```
        LD  IX,BLK2 + 2     INITIALIZE START OF SOURCE
        LD  IY,BLK1 + 2     INITIALIZE START OF DEST
NXT1    LD  B,(IX)          SAME AS (IX + 0)
        LD  (IY),B          SAME AS (IY + 0)
        DEC IX              POINT TO NEXT BYTE SOURCE
        DEC IY              POINT TO NEXT BYTE DEST
NXT2    LD  B,(IX)          NEXT
        LD  (IY),B
        DEC IX
        DEC IY
NXT3    LD  B,(IX)          NEXT
        LD  (IY),B
```

Code such as the above is inefficient in memory storage because the same basic operation is repeated many times. The transfers at NXT1, NXT2, and NXT3 are almost identical. If 100 bytes were to be transferred, it would of course be ludicrous to repeat the identical actions 100 times. The most efficient way to implement repetitive actions is by *looping* back to the same set of instructions for as many times, N, as required. This is done in the following program which uses IX and IY as source and destination pointers as before and moves 100 bytes from BLK2 to BLK1. The initial count, N = 100, is held in the C register and is decremented down to 0. The loop at




```
        LD  IX,BLK2     STRT OF SRC BLK
        LD  IY,BLK1     STRT OF DST BLK

        LD  C,100       SET N = 100
        LD  B,(IX)      LOAD BYTE
        LD  (IY),B      STORE BYTE
        INC IX          BMP IND BY ONE

        INC IY
        DEC C           N - 1
        JP  NZ,LOOP     GO IF NT DN (Z=1)
                        DONE HERE
```

LOOP is executed 100 times as long as N = 100 to 1, the Z flag is not set and the conditional branch JP NZ,LOOP is made. IX and IY are incremented by one each time through the loop to point to the next position in the blocks.

The values of IX, IY, C, and B for the first 5 and last 5 *iterations* of the loop are shown in Fig. 10-2.

IX	IY	C	B	
BLK2	BLK1	100	-	INITIALIZATION
BLK2+1	BLK1+1	99	BYTE 1	ITERATION 1 (AFTER)
BLK2+2	BLK1+2	98	2	2
BLK2+3	BLK1+3	97	3	3
BLK2+4	BLK1+4	96	4	4
BLK2+5	BLK1+5	95	BYTE 5	5
≈	≈			≈
BLK2+95	BLK1+95	4	BYTE 96	ITERATION 96 (AFTER)
BLK2+96	BLK1+96	3	97	97
BLK2+97	BLK1+97	2	98	98
BLK2+98	BLK1+98	1	99	99
BLK2+99	BLK1+99	0	BYTE 100	100

Fig. 10-2. Indexing example.

8-BIT MOVES USING THE A REGISTER AND EXTENDED ADDRESSING

The A register can be loaded or stored using extended addressing. In this case, the address specified is in the instruction itself, and completely random addressing can be done without the need for

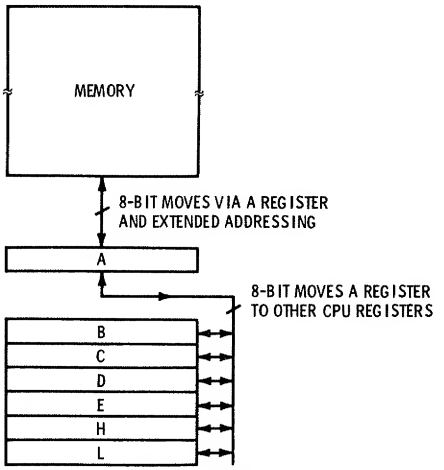


Fig. 10-3. A register used for random addressing.

setting up any pointers or index registers. This instruction is probably the one most frequently used for moving eight bits of data into CPU registers and for storing CPU register data via the A register, as shown in Fig. 10-3. The A register is the path for all of the other CPU registers in this case.

The following routine loads A, B, C, and D with VAR1, VAR2, VAR3, and VAR4 after first storing the registers in STRA, STRB, STRC, and STRD using this kind of addressing.

```
LD (STRA),A          STORE A
LD A,B
LD (STRB),A          STORE B
LD A,C
LD (STRC),A          STORE C
LD A,D
LD (STRD),A          STORE D
LD A,(VAR4)          GET VAR4 FOR D
LD D,A
LD A,(VAR3)          GET VAR3 FOR C
LD C,A
LD A,(VAR2)          GET VAR2 FOR B
LD B,A
LD A,(VAR1)          GET VAR1 FOR A
```

8-BIT MOVES USING THE A REGISTER AND BC OR DE REGISTER INDIRECT

The four instructions LD A,(DE); LD A,(BC); LD (DE),A; and LD (BC),A use BC or DE as pointers in a manner similar to the way HL is used as a pointer for the previously discussed moves. Here again, this addressing mode is very efficient as long as the data being accessed is contiguous data in a block or table. A few examples ago, the use of the index registers for moving data from one block to another was presented. The following routine does the same, and it can be seen that the actions are virtually identical. BC

```
        LD  BC,BLK2      START OF SOURCE BLOCK
        LD  DE,BLK1      START OF DEST BLOCK
        LD  L,100        SET N = 100
AGAIN   LD  A,(BC)       LOAD BYTE
        LD  (DE),A       STORE BYTE
        INC BC           BUMP INDICES BY ONE
        INC DE
        DEC L            N - 1
        JP  NZ,AGAIN     GO IF NOT DONE (Z = 1)
DONE
```

points to the source block, DE points to the destination block, and L contains the count, 100 in this case.

16-BIT MOVES

Data movement discussed above involved moving eight bits at a time. The Z-80 has many instructions to move data two bytes, or sixteen bits at a time, however. Data moved in this width are loaded or stored between register pairs BC, DE, and HL; registers SP, IX, and IY and memory. Sixteen-bit data operations allow the following:

1. Immediate loads of BC, DE, HL, SP, IX, IY
2. Transferring data *from* memory *to* BC, DE, HL, SP, IX, or IY, or the reverse
3. Transferring data from the HL, IX, or IY to SP
4. Pushing and popping BC, DE, HL, AF, IX, or IY to the stack

Many of the loads of the register pairs, SP, or index registers will, of course, involve loads of memory addresses. Sixteen bits will hold all 64-K external-memory addresses for the Z-80, and the instructions in this group have specifically been set up for handling address-related data. If convenient, though, all instructions can be used to load and store nonaddress operands, such as 16-bit double-precision values or ASCII character data.

IMMEDIATE LOADS OF 16 BITS

Many of the immediate loads have previously been illustrated in this chapter. BC, DE, HL, IX, and IY are typically loaded with the starting address of data blocks containing data to be processed as in:

```
        LD   BC,DATA1     LOAD ADDRESS OF DATA 1
        LD   DE,DATA2     LOAD ADDRESS OF DATA 2
        LD   HL,DATA3     LOAD ADDRESS OF DATA 3
        LD   IX,DATA4     LOAD ADDRESS OF DATA 4
        LD   IY,DATA5     LOAD ADDRESS OF DATA 5
         ↘
DATA1   DEFS 100          DATA BLOCK OF 100 BYTES
DATA2   DEFS 50           DATA BLOCK OF  50 BYTES
DATA3   DEFS 20           DATA BLOCK OF  20 BYTES
DATA4   DEFS 60           DATA BLOCK OF  60 BYTES
DATA5   DEFS 100          DATA BLOCK OF 100 BYTES
```

The stack pointer register, SP, almost always points to the area of memory allocated as the stack area, however, and not to a predefined data block. The SP is *initialized* to an address value that represents the *top of stack* by an LD SP,NN instruction. Since the SP always

points to the *last used* location in the stack and is *decremented* before storage of data is performed, the SP must be loaded with an address value corresponding to one greater than the first location to be used as the stack. If, for example, stack storage is to occupy 100H bytes from address 3FFFH down to 3F00, the SP would be initialized as follows:

```
0100            LD   SP,4000H      LOAD TOP OF STACK
                     OR
                LD   SP,TOPS
                 ↘
3F00            DEFS 100H          DEFINE STACK AREA
4000    TOPS    EQU  $               OF 256 BYTES
```

Subsequent pushes to the stack (there can be no pops as there has been no data storage in the stack at this point) will decrement the SP by one before storage. The first byte of data will be stored at 3FFF, the next at 3FFE, and so forth.

16-BIT TRANSFERS TO AND FROM MEMORY

The BC, DE, HL, SP, IX, IY, or SP registers may be loaded from or stored to memory by instructions in this group. As an example, suppose that the BC, DE, and HL registers are to be loaded with the addresses of three blocks of memory, but their contents are to be saved and restored for later use. As an alternative to storage in the stack (covered a little later in this chapter), the three register pairs may be saved by:

```
                LD   (SAVB),BC     SAVE BC
                LD   (SAVD),DE     SAVE DE
                LD   (SAVH),HL     SAVE HL
                 ↘
SAVB    DEFS 2                     STORAGE FOR BC
SAVD    DEFS 2                     STORAGE FOR DE
SAVH    DEFS 2                     STORAGE FOR HL
```

Notice that the storage locations reserved for each of the register pairs must be *two* bytes. Later, when the register pairs are to be reloaded with their original values, the instructions below may be executed:

```
LD   BC,(SAVB)     RESTORE BC
LD   DE,(SAVD)     RESTORE DE
LD   HL,(SAVL)     RESTORE HL
```

As in many groups of instructions, the format of the assembly language arguments is extremely important. In the following code, LD

HL,SAVL loads the *address* of SAVL (1000H) while LD HL, (SAVL) loads the *contents* of SAVL.

```
                LD    HL,SAVL       LOADS 1000H
                 ↓
                LD    HL,(SAVH)     LOADS 123AH
                 ↓
1000    SAVL    DEFW  1234H         CONTENTS OF 1000H
                                      IS 1234H
```

16-BIT DATA TRANSFERS TO THE STACK

The Z-80 allows the transfer of data from the HL, IX, and IY registers to the stack pointer register, but not the reverse. Examples of these transfers are:

```
LD SP,HL        HL TO SP
 ↓
LD SP,IX        IX TO SP
 ↓
LD SP,IY        IY TO SP
```

16-BIT STACK OPERATIONS

The title of this subsection is a misnomer, for *all* stack operations involve the transfer of sixteen bits or two bytes of data at a time. Eight bits cannot be pushed or popped to the stack as in other microcomputers. This is not a great disadvantage, although it may create a little more overhead when only one register is to be saved in the stack for temporary storage. In the Z-80 register pairs BC, DE, HL, AF, and registers IX and IY may be pushed and popped to the memory stack. As each is pushed to the stack, the data in the high-order byte of the register pair is put into the top of stack −1 and the data in the low-order byte is put into top of stack −2. The SP register is decremented by one before each byte is pushed. The following explains stack action on a push of a register pair, IX or IY.

```
LD   SP,1000H      INITIALIZE SP TO 1000H
 ↓
PUSH AF            A TO 0FFFH, F TO 0FFEH
PUSH BC            B TO 0FFDH, C TO 0FFCH
PUSH DE            D TO 0FFBH, E TO 0FFAH
PUSH HL            H TO 0FF9H, L TO 0FF8H
PUSH IX            IX15-8 TO 0FF7H, IX7-0 TO 0FF6H
PUSH IY            IY15-8 TO 0FF5H, IY7-0 TO 0FF4H
```

As the reader would suspect, the F(lags) register is treated as an 8-bit lower-order register on stack operations.

As data is popped from the stack, the process is reversed. The

low-order byte is pulled from the top of stack and put into the F, C, E, L, IX_{low}, or IY_{low} registers, the SP is *then* incremented and the high-order byte is put into the high-order register of the register pair or higher-order byte of the IX or IY registers.

Stack storage is employed for the following reasons:

1. Storage of the environment during interrupt processing
2. Temporary storage of CPU registers
3. As a way to transfer data between CPU registers
4. Subroutine use

Stack operations during interrupt actions and subroutine use will be discussed later. The other two uses are somewhat obvious. At any time, data from one of the register pairs, IX or IY, may be saved in the stack by execution of a PUSH instruction. Later the data may be retrieved by a POP instruction. There is no condition that states that the data POPped must be restored to the same register pair and the stack may therefore conveniently be used to transfer data between registers, as in the following example which *exchanges* the BC and IY, and DE and IX registers.

```
PUSH BC          STACK NOW HAS BC
PUSH IY          STACK NOW HAS BC, IY
PUSH DE          NOW BC, IY, DE
PUSH IX          NOW BC, IY, DE, IX
POP  DE          IX TO DE
POP  IX          DE TO IX
POP  BC          IY TO BC
POP  IY          BC TO IY
```

The stack register may also be used to facilitate processing of strings of data, although care must be taken to maintain the stack pointer properly when this is done. As an example of this, suppose that locations 177FH through 1700H had a string of ASCII characters with the first character in 1700H and the last in 177FH. (Data can easily be stored in this fashion by use of the increment type instructions.) The following code processes each of the characters, one at a time, *providing that the stack is not used for any other storage anywhere in the processing.* This means that no maskable or nonmaskable interrupts may occur or that no other routines that use the stack may be employed during the time the processing occurs.

```
    LD   (SAVP),SP     SAVE CURRENT STACK POINTER
    LD   SP,1700H      INITIALIZE SP TO DATA
┌─→POP BC              FIRST BYTE IN C, NEXT IN B
│    ↻
│   (PROCESS)          (LOOP HERE 128 TIMES)
└──┘
    LD   SP,(SAVP)     RESTORE SP TO ORIGINAL STACK
```

Although the processing above will certainly work, it is probably best to process the string data by other means, especially since interrupts will cause catastrophic results. The block instructions implemented in the Z-80 will permit processing of string data in a much cleaner fashion.

BLOCK TRANSFER INSTRUCTIONS

The Block Transfer instructions in the Z-80 offer a means to move up to 64-K bytes of data automatically or semi-automatically from one area of memory to another. The ground rule for moving data is the following:

NEVER MOVE LARGE BLOCKS OF DATA FROM ONE AREA OF MEMORY TO ANOTHER UNLESS UNAVOIDABLE!

There are many ways to avoid large data movements. Data should be input or output directly to a buffer in which they can be processed. Tables can be set up properly to avoid reformatting of data. Programming structures such as linked lists may be employed instead of contiguous tables. The primary reason for avoiding block-data transfers is the enormous amount of time that they require. To move 1000 bytes of data at 10 microseconds per byte requires 10 milliseconds or 1/100 of a second. Although the time required per byte in the Z-80 is about one third of this, block movements still take large amounts of time in comparison to other program operations.

With the above proviso in mind, let us see how the block transfer instructions in the Z-80 can be set up. The first of these is the LDI instruction. The LDI requires that the HL register pair points to the source data block and that the DE register pair points to the destination data block. The BC register pair contains a byte count. To transfer 100H bytes of data from a data block starting at location 1000H to a data block starting at 2000H, the following code could be employed.

```
LD HL,1000H     SET UP SOURCE PNTR
LD DE,2000H     SET UP DEST PNTR
LD BC,100H      100 BYTES
```

After initialization, each time an LDI was executed a byte would be transferred *from* the location pointed to by the HL *to* the location pointed to by DE. The byte count in BC would then be decremented by one. When the byte count reached zero, the *P/V* flag would be reset. The P/V flag therefore can be tested to determine the end of the transfer. The following code transfers the data:

```
        LD  HL,1000H    SET UP SOURCE PNTR
        LD  DE,2000H    SET UP DEST PNTR
        LD  BC,100H     100 BYTES
LOOP    LDI             TRANSFER BYTE
        JP  PE,LOOP     GO IF NOT DONE
DONE    (
```

Note that the P/V flag is *set* if the byte count is not equal to 0. This is equivalent to "parity even" or PE. The jump will be performed as long as P/V equals 1 and 100 bytes have not been transferred.

This block transfer instruction is "semi-automatic" compared to the LDIR that transfers the specified number of bytes in BC automatically in one instruction. What is the advantage in having something other than a fully automatic block transfer? One obvious advantage is that the LDI allows intermediate processing to occur between the transfer and the jump back to the next transfer. Suppose that the data must not only be moved, but that the movement be terminated on zero data. Thus, a maximum of N bytes would be moved; however, if any of the source bytes were 0 the move would stop. The following code terminates the move if the next byte to be moved is zero. The source byte about to be moved is first tested before the move occurs, and if zero, the move is terminated. The OR A,A, instruction tests the zero/nonzero status of the byte without affecting the byte. The Z flag is reset if any bit in the byte is a one and set if all bits are zeros.

```
MOVE    LD  HL,1000H    SET UP SOURCE PNTR
        LD  DE,2000H    SET UP DEST PNTR
        LD  BC,100H     100 BYTES MAXIMUM
NEXT    LDI             TRANSFER BYTE
        JP  PO,DONE     GO IF DONE (MAXIMUM)
        LD  A,(HL)      GET NEXT BYTE
        OR  A           TEST BYTE FOR ZERO
        JP  NZ,NEXT     CONTINUE IF NOT ZERO
DONE    (
```

Another advantage of the LDI is that it can be used to move noncontiguous data. Suppose that there is a table of data that is 100H bytes long and that every fourth byte is to be moved to a new data area as shown in Fig. 10-4. The number of transfers must be 256/4 or 64 and the new storage area will hold the source bytes as shown. The following code moves the data:

```
        LD  HL,SRTAB    SET UP SOURCE PNTR
        LD  DE,DSTTB    SET UP DEST PNTR
        LD  BC,100H/4   SET UP BYTE COUNT
```

```
NEXT    LDI              TRANSFER BYTE
        INC HL           POINT TO NEXT BYTE
        INC HL
        INC HL
        JP  PE,NEXT      GO IF NOT DONE
DONE    ⟨
```

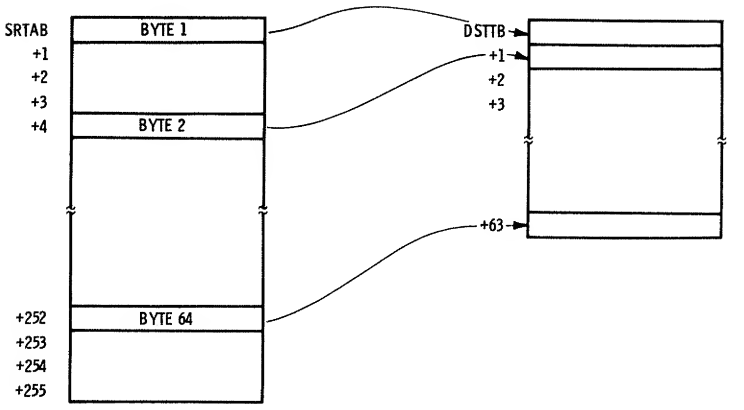


Fig. 10-4. Moving noncontiguous data with LDI.

There are several subtleties in the above code. The expression 100H/4 will work in many assemblers and enables an *assembly-time* calculation of the number of bytes. After the LDI has transferred the Ith byte, the HL register points to I + 1. The three increments bump the HL to point to I + 4.

If no processing is to take place between the transfer of individual bytes, then the LDIR may be used. The LDIR is set up in exactly the same manner as the LDI. If N bytes are to be transferred, however, the LDIR will execute N times. For each transfer, the LDIR takes 5.25 microseconds (the LDI takes 4.0 microseconds) except for the last transfer (BC = 0) in which the LDIR takes 4.0 microseconds.

```
        LD   HL,STRTS    SOURCE START
        LD   DE,STRTD    DEST START
        LD   BC,64       # OF BYTES
        LDIR             TRANSFER 64 BYTES IN
DONE    ⟨                  ABOUT 335 MICROSEC.
```

In the LDI and LDIR instructions, data is transferred in forward order, that is, it is transferred starting from low memory and ascending to high memory. The only difference between the LDI and

LDIR and the LDD and LDDR is that the latter two transfer data in descending order. HL and DE are set to the *ending* address of the source and destination data blocks, respectively, and each is decremented to point to the next lower byte by the LDD or LDDR. To transfer data from the previous example, the code would read:

```
        LD    HL,ENDS       SOURCE END
        LD    DE,ENDD       DEST END
        LD    BC,64         # OF BYTES
        LDDR                TRANSFER 64 BYTES
DONE    )
        (
        \/
STAB    DEFS  64            SOURCE TABLE
ENDS    EQU   $-1
DTAB    DEFS  64            DESTINATION TABLE
ENDD    EQU   $-1
```

EXCHANGE GROUP

There are six instructions in the exchange group. Two of them transfer data between the current set of CPU registers and the primed (′) set. Three others allow the HL and index registers to exchange their contents with the top of the stack. The last simply exchanges the contents of DE with HL.

When the CPU is initialized, one set of the two eight-register sets becomes the current set. The other set containing A′, F′, B′, C′, D′, E′, H′, and L′ may be accessed via the two exchange instructions EX AF,AF′ and EXX. EX AF,AF′ swaps the contents of A and F with A′ and F′. To temporarily store A and F, the following code could be used:

```
EX  AF,AF'              SAVE A,F
    (
    \/
PROCESSING
    (
    \/
EX  AF,AF'              RESTORE A,F
```

Likewise, EXX swaps BC, DE, and HL with BC′, DE′, and HL′.

```
EXX                     SAVE BC, DE, HL
LD  BC,NEW1             NEW ADDRESSES
LD  DE,NEW2
LD  HL,NEW3
    (
    \/
PROCESSING
    (
    \/
EXX                     RESTORE BC, DE,HL
```

EX AF,AF′ and EXX would probably be used most frequently in saving the environment during interrupt processing. One reason for not using the instructions at other times is that if the primed registers are used for temporary storage and also used for interrupt storage, it is very probable that some of the temporary data will be destroyed if interrupts are permitted while both sets of CPU registers are being used. It is best to reserve the primed registers for processing use only and utilize the stack or memory for temporary storage.

The EX DE,HL instruction swaps the contents of register pair DE and HL. The instruction is useful for moving data from the DE to HL for the limited arithmetic operations that can be performed to HL. As an example of this, suppose the contents of DE were to be doubled. The following code would move DE to HL, add HL to itself to double the contents and move the result back into DE.

```
EX   DE,HL        DE TO HL
ADD HL,HL         HL + HL TO HL
EX   DE,HL        HL TO DE
```

The remaining three instructions in this group exchange the contents of the top of stack with either HL, IX, or IY. The SP is *not affected* by the swap. Clearly, the manufacturer had a good reason for the exchange of HL and top of stack [EX (SP),HL – Intel] and the index registers [EX (SP), IX or . . . . IY – Zilog]. It will be left as an exercise for the reader to discover for himself what those reasons are. (Seriously, one application is to permit adjustment of the return address for a call to enable a return to a location other than the one following the call.)

CHAPTER 11

Arithmetic and Logical Operations— 8-and 16-Bit Arithmetic Group, Decimal Arithmetic

The arithmetic and logical operations covered in this chapter include adds, subtracts, logical ORs, ANDs, exclusive ORs, compares, increments, and decrements. All of these operations can be performed in 8-bit precision and the adds, subtracts, increments, and decrements can also be performed in 16-bit precision using register pairs. The functions performed in these groups are some of the most basic operations that a computer can perform. Additionally, the Z-80 allows bcd or decimal adds and subtracts by means of a special "decimal adjust."

8-BIT ARITHMETIC OPERATIONS

In 8-bit arithmetic operations, two 8-bit operands are added or subtracted. One of the operands must be in the A register while the other operand may be an immediate operand, an operand in another CPU register, or an operand from memory. The result of the operation always goes to the A register. The add or subtract function sets the condition-code flags in the flag register on the result of the operation as discussed in Chapter 6. A variety of addressing modes may be used to fetch the second operand, including register indirect and indexing addressing.

The simplest operation in this group is an 8-bit add. If a checksum of a block of 63 bytes was to be computed, the following routine

would add together all 63 bytes after clearing the A register. The checksum is then stored at the beginning of the block in location BLK. (A checksum such as this would be used for comparison purposes on subsequent retrievals of the data block; by repeating the add and comparing the result with the inherent checksum, the validity of the data could be established.)

```
        SUB   A          A - A CLEARS A
        LD    IX,BLK+63  SET PNTR TO END OF DATA
        LD    B,63       SET COUNT TO 63
LOOP    ADD   A,(IX)     ADD NEXT BYTE
        DEC   IX         INDEX - 1
        DEC   B          COUNT - 1
        JP    NZ,LOOP    CONTINUE IF NOT 63 BYTES
DONE    LD    (IX),A     STORE A IN BLK
        ↓
BLK     DEFS  64         CHECKSUM + 63 DATA BYTES
```

In the above program, the last instruction LD (IX),A stored the checksum held in A to the location pointed to by the contents of the index register IX. As the data started at BLK+63 and ended at BLK+1, the index register pointed to BLK+0 after the last iteration of the loop and the checksum could be stored without further adjustment to the index register.

If the block of data were to be read in from the I/O device and the checksum to be calculated and compared, a subtract instruction could be used to advantage.

```
        LD    IX,INBLK+1 SET PNTR TO START OF DATA
        LD    A,(IX-1)   GET CHECKSUM
        LD    B,63       SET COUNT TO 63
LOOP    SUB   (IX)       SUBTRACT NEXT BYTE
        INC   IX         POINT TO NEXT BYTE
        DEC   B          COUNT - 1
        JP    NZ,LOOP    CONTINUE IF NOT 63 BYTES
DONE    OR    A          TEST CONTENTS OF A
        JP    NZ,ERROR   GO IF ERROR IN DATA
NERROR  ~                NO ERROR
        ~
        ~
ERROR   ↓
```

The IX register is first set to the start of the data at INBLK+1. The next instruction loads the checksum byte at INBLK (IX−1 is (INBLK+1) − 1 = INBLK) into A. Then the data at INBLK+1 through INBLK+63 is subtracted from the partial checksum in A. At the end (DONE), the contents of A should be 0 if the data is valid. An OR is done which simply serves to set the flags for the conditional branch JP NZ,ERROR. If A is not zero, location ERROR is

executed, presumably an error routine, otherwise the next instruction at NERROR is executed.

When two 8-bit operands are added or subtracted the sign, zero, parity, and carry flags are affected. Examples for use of the zero flag have been presented previously. The sign flag may be tested by a conditional jump on P(ositive) or M(inus). The following example tests for an ASCII character between 30H and 39H (decimal 0 through 9).

```
        LD    A,(CHAR)    GET ASCII CHARACTER
TEST09  SUB   30H         SUBTRACT 30
        JP    M,ERROR     GO IF LESS THAN 30
        SUB   10          SUBTRACT 10
        JP    P,ERROR     GO IF 3A OR GREATER
OK      ADD   10          THIS CHARACTER 30 TO 39
```

In this somewhat inefficient test (a compare is called for in place of the SUB 10), the ASCII character is loaded and an immediate 30H is subtracted from the character. If the character is less than ASCII 30H (decimal 0), the result is negative, the sign flag is set, and a jump to ERROR is taken. If the character is greater or equal to 30H, 10 is subtracted from the first result yielding a negative number for all valid ASCII characters (now 0 through 9) or a positive number for all ASCII characters greater than ASCII 39H (decimal 9). A test at the JP causes a jump to the error routine if this *limit check* fails. Finally, the decimal equivalent of the ASCII character is restored by adding 10 to yield 0 – 9 for the converted character.

If an add or subtract results in an effective add of two 8-bit operands of similar signs, overflow is possible and can be tested by a conditional branch on the P/V flag. Overflow will occur and the P/V flag will be set if the result exceeds −128 (80H) or +127 (7FH). The following code tests for overflow and effects a jump to an error routine if overflow has occurred.

```
        LD    A,(OPND1)   LOAD OPERAND 1
        LD    B,A         INTO B
        LD    A,(OPND2)   LOAD OPERAND 2
        ADD   A,B         ADD OPND 2 TO OPND 1
        JP    PE,ERROR    JUMP IF OVERFLOW
NERROR  (                 NO OVERFLOW HERE
        ↓
```

The carry flag finds most use during double-precision or multiple-precision operations. If the required precision is 16 bits, many operations can be implemented by the 16-bit arithmetic instructions discussed later in this section. For the general case, however, where the precision may exceed 16 bits, the Z-80 has addition and subtraction instructions that make use of the carry and allow operands to

be any size, one byte to n bytes. If an operand of four bytes or 32 bits is required, for example, numbers from 0 to 4,294,967,296 may be handled (or the equivalent range of negative and positive numbers). To add or subtract two four-byte operands, a carry or borrow must be *propagated* to the higher-order bytes. This means that an add or subtract to the three higher orders will not suffice; there must be an add or subtract *with carry* (carry also represents a borrow in the Z-80 and other machines). The following code performs a four-byte add and subtract on two four-byte operands located in OP1 and OP2. OP2 is added or subtracted to OP1 and the result put in OP1. The first add or subtract is of the lowest order and no carry or borrow exists from previous orders, therefore, an ADD or SUB is used. Subsequent adds and subtracts utilize the ADC (add with carry) and SBC [subtract with carry (borrow)] instructions to propagate the carry or borrow.

```
ADD4    LD    IX,OP1+3     POINT TO LOW-ORDER BYTE
        LD    IY,OP2+3     POINT TO LOW-ORDER BYTE
        LD    A,(IX)
        ADD   A,(IY)       OP1 + OP2 BYTE 3
        LD    (IX),A       STORE RESULT IN OP1+3
        LD    A,(IX-1)
        ADC   A,(IY-1)     OP1 + OP2 BYTE 2
        LD    (IX-1),A     STORE RESULT IN OP1 + 2
        LD    A,(IX-2)
        ADC   A,(IY-2)     OP1 + OP2 BYTE 1
        LD    (IX-2),A     STORE RESULT IN OP1 + 1
        LD    A,(IX-3)
        ADC   A,(IY-3)     OP1 + OP2 BYTE 0
        LD    (IX-3),A     STORE RESULT IN OP1
DONE    (
        v
```

```
SUB4    LD    IX,OP1+3     POINT TO LOW-ORDER BYTE
        LD    IY,OP2+3     POINT TO LOW-ORDER BYTE
        LD    B,4          INITIALIZE COUNT
        XOR   A            CLEAR CARRY
LOOP4   LD    A,(IX)       LOAD BYTE
        SBC   (IY)         OP1 - OP2
        LD    (IX),A       STORE RESULT
        DEC   IX           POINT TO NEXT HIGH-ORDER
        DEC   IY
        DEC   B            DECREMENT COUNT
        JP    NZ,LOOP4     GO IF NOT DONE
DONE    (
        v
```

The examples of add and subtract illustrate two different approaches to the solution of the same problem. The add example

utilizes a *linear* or *in-line* approach while the subtract example is an iterative approach using a loop. In the add, a separate load, add, and store is performed for each of the four bytes. All but the first add adds in the carry from the lower-order byte by an ADC instruction. The subtract example has some subtleties in it. The index registers are initialized to the low-order address of the operands. A count of four for the four subtracts is set up in the B register. For the first add, the carry must be cleared and an XOR instruction is used to accomplish the clear. An XOR always clears the carry. Now the first operands are subtracted and the result stored in the low-order byte of the destination operand. The IX and IY registers are decremented by one to point to OP1+2 and OP2+2. The count in the B register is decremented and, because it is not yet zero, the jump is taken to LOOP4. On the next subtract, the carry will be set, or reset, dependent on the last SBC instruction *since no other instruction in the loop affects the carry.* After four subtracts from low to high order, the count in B is 0 and the instruction at DONE is executed and the result is in OP1 to OP1+3.

8-BIT LOGICAL OPERATIONS

The 8-bit logical operations are similar to the 8-bit adds in that the same addressing modes are permitted and the A register contains the primary operand and holds the result at the end of instruction execution. The three logical operations that can be performed are the logical AND, OR, and exclusive OR. The rules for these logical operations are shown in Table 11-1.

Table 11-1. Logical Operations

Instruction	Logical Operation	Symbol
AND	0 ∧ 0 = 0; 0 ∧ 1 = 0; 1 ∧ 0 = 0; 1 ∧ 1 = 1	∧ is the symbol for AND
OR	0 ∨ 0 = 0; 0 ∨ 1 = 1; 1 ∨ 0 = 1; 1 ∨ 1 = 1	∨ is the symbol for OR
XOR	0 ⊕ 0 = 0; 0 ⊕ 1 = 1; 1 ⊕ 0 = 1; 1 ⊕ 1 = 0	⊕ is the symbol for exclusive OR

Each logical operation is done for every bit position on a bit-by-bit basis. One bit position does not affect any other bit position and consequently there can be no carry.

The AND instruction can be used to mask in, or mask out, unwanted *fields* within data bytes. Suppose, for example, that an 8-bit byte in memory holds two *packed* bcd digits, one in bits 7-4 and the other in bits 3-0. It is necessary to test the second bcd digit. The first bcd digit would be masked out and the second bcd digit would remain for testing by the following code.

```
        LD    A,(DIGITS)   GET 2 BCD DIGITS
        AND   OFH          MASK OUT HIGH ORDER
TEST    ⟨
        ↘
```

Since bits 7-4 of the immediate data value were 0, the corresponding bits of the result in the A register can never be one. As bits 3-0 of the immediate data value were ones, however, all bits of the low-order bcd digit "fall-through" to the result. If the data at DIGITS was 37H, the result after the AND instruction was executed would be $37H \wedge 0FH = 07H$.

The OR instruction is used to *merge* data into a field or to unconditionally set certain bits within a data byte. If one bcd digit was in the A register in the form $0000JJJJ_2$ and the second was in the B register in the form $KKKK0000_2$, a merged result of the form $KKKKJJJJ_2$ could be obtained by:

```
        OR    B      MERGE TWO BCD DIGITS
```

As another example of the ORing function, suppose that the highest order, or most significant bit, in a table of ten bytes was to be unconditionally set. The following code would set the msb of each of the ten bytes without affecting the remainder of the byte. Note that an ADD of 80H would not necessarily do the same thing as adding 80H to values of 80H to FFH would *reset* the msb.

```
        LD    IX,TABLE     SET UP INDEX
        LD    B,10         SET UP COUNT
LOOP    LD    A,(IX)       GET BYTE FROM TABLE
        OR    80H          SET MSB
        LD    (IX),A       STORE BYTE BACK IN TABLE
        INC   IX           POINT TO NEXT BYTE
        DCR   B            DECREMENT COUNT
        JP    NZ,LOOP      JUMP IF NOT DONE
DONE    ⟨
        ↘
```

The exclusive OR instruction is not used as frequently as the AND and OR instructions. One use is to "toggle" a bit between a one and a zero, either for timing or for maintaining a count of two. The following instructions allow a loop starting at LOOP to be reentered twice only:

```
        LD      A,0             SET COUNT TO 0
LOOP    ⌇
        ↓
        (PROCESSING)

        XOR     1               TOGGLE COUNT
        JP      NZ,LOOP         GO IF 1 FOR SECOND PASS
DONE    ⌇                       DONE WITH TWO PASSES
        ↓
```

8-BIT COMPARES

A compare operation is functionally similar to a subtract except that the result of the comparison does not replace the source operand. Only the condition-code flags are set on the result of the comparison. A compare can therefore be used to test one operand against another and a following conditional jump can be made on the results of the comparison. As in the add or subtract, the 8-bit compare can use a variety of addressing modes including register indirect and indexed addressing.

As an example of a compare, let us look at the following program which finds the smallest number in a list of *positive* numbers. As each new number is accessed, it is compared with the previous smallest number. If the new entry is smaller, it replaces the previous smallest number in the B register. When the last number in the list has been compared, the B register holds the smallest number in the list. The numbers in the list may range from 0 to +127 in *unsorted*

```
GTSMLL  LDX     IX,LIST         SET UP LIST ADDRESS
        LD      C,-1            STORE TERMINATOR
        LD      B,127           INITIALIZE SMALLEST NUMBER
NEXT    LD      A,(IX)          GET NEXT ENTRY
        CP      C               COMPARE TO -1
        JR      Z,DONE          GO IF AT END OF LIST
        INC     IX              POINT TO NEXT NUMBER
        CP      B               COMPARE NEW-SMALLEST
        JP      P,NEXT          GO IF NEW >=SMALLEST
        LD      B,A             NEW TO SMALLEST
        JP      NEXT            CONTINUE
DONE    ⌇                       SMALLEST IN B
        ↓
LIST    DEFB    20              LISTOF NUMBERS
        DEFB    32              UNSORTED
        DEFB    1               MAY BE 0-127
        DEFB    0
        DEFB    37
        DEFB    112
        DEFB    3
        DEFB    -1              TERMINATOR
```

or random order. Because only positive numbers are represented, a *magnitude* compare rather than an *algebraic* compare (positive and negative numbers) may be performed. Because no negative numbers are allowed, a −1 is used to terminate the list. This makes the check for the end of the list somewhat easier than decrementing a count as the current number may be tested for a −1 state before the smallest comparison is made.

The index register is set up with the address of the list in the first instruction. Next, the terminator value −1 is stored in C for each register comparison and the smallest number in B is initialized to 127_{10}. Since 127 is the largest permissable value, all of the numbers in the list must be less than or equal to 127. The list shown is generated at assembly-time, but a list generated *dynamically* could just as easily be used. As each entry in the list is picked up, it is compared to −1 and if equal to (Z), a *relative branch* is made to DONE. The assembler will automatically fill in the proper displacement in the JR instruction to cause a branch to DONE. Next, the index register is incremented in preparation for the next comparison. A comparison is then done of the current value to the smallest in B; if the current value is smaller, it replaces the contents of B. A jump to NEXT is then made for the next comparison.

The above example is of a magnitude or unsigned comparison. How is the signed comparison implemented? There are four cases in the comparison of signed numbers, comparison of a ++, +−, −+, and −−. If the signs of the operands are the same, there will be a carry if the contents of A are greater than or equal to the second operand. If the signs of the operands are different, there will be a carry if the contents of A are less than 0. The following routine performs an algebraic compare on two operands, either of which may be −128 to +127. Jumps are made on three equality combinations <, =, or > based on a comparison of A:B (A−B).

```
CMPARE   CP    B        A:B
         JP    Z,EQUAL  GO IF A=B
         PUSH  AF       A AND FLAGS
         XOR   B
         JP    P,SAME   GO IF SIGNS THE SAME
         POP   AF       RESTORE A, FLAGS
TEST     JP    C,LESST  A<B
         JP    GREAT    A>B
SAME     POP   AF       RESTORE A, FLAGS
         CCF            COMPLEMENT CARRY FOR TEST
         JP    TEST
```

The program above first tests the equality of A and B. If they are equal, a jump is made to equal. Then the flags and A are saved in

the stack and an XOR is done to test the sense of the two operands. If the sign flag is set after the XOR, the signs are different. If the signs are the same, A and the flags are restored. The CY flag holds the results of the previous comparison which is complemented by CCF for the test at TEST. The branch to LESST is taken if (+):(+) or (−):(−) and there is no carry, or if (+):(−) or (−):(+) and there is a carry. The branch to GREAT is taken for the inverse conditions.

8-BIT INCREMENT AND DECREMENT

The increment instruction INC and decrement instruction DEC can be used to increment or decrement contents of a CPU register or memory location by one. Any of the general-purpose 8-bit registers A, B, C, D, E, H, and L may be modified. Note that the associated register of the register pair is not affected. When a memory location is modified, register indirect addressing using the HL register or indexed addressing may be used. The instructions are straightforward and should hold no surprises for the programmer.

16-BIT ARITHMETIC OPERATIONS

Just as the A register was the main accumulator that was used for 8-bit arithmetic and logical operations, the HL register is used as an accumulator for 16-bit arithmetic operations. Register pairs BC, DE, HL, and SP may be added to, or subtracted from, the contents of the HL register. The add may be with or without carry, but the subtract is always with borrow. The Z-80 instruction also allows BC, DE, HL, or SP to be added to the contents of IX or IY in a simple add without carry, and allows any register BC, DE, HL, SP, IX, or IY to be incremented or decremented by one in a 16-bit operation.

In the 8-bit examples, two programs performing four-byte addition and subtraction were listed. The 16-bit instructions offer an alternative way to implement the problem. If the four-byte operands are at OP1 and OP2, the following two programs compute OP1 + OP2 and OP1 − OP2 and place the four-byte result in OP1.

```
ADD4    LD    HL,(OP1+2)     GET TWO LS BYTES OP1
        LD    BC,(OP2+2)     GET TWO LS BYTES OP2
        ADD   HL,BC          BC + HL TO HL
        LD    (OP1+2),HL     STORE LS BYTES OF RESULT
        LD    HL,(OP1)       GET TWO MS BYTES OP1
        LD    BC,(OP2)       GET TWO MS BYTES OP2
        ADC   HL,BC          ADD WITH CARRY
        LD    (OP1),HL       STORE MS BYTES OF RESULT
DONE    ↓
```

```
SUB4    LD    HL,(OP1+2)     GET TWO LS BYTES OP1
        LD    DE,(OP2+2)     GET TWO LS BYTES OP2
        OR    A              RESET CARRY
        SBC   HL,DE          HL - DE TO HL
        LD    (OP1+2),HL     STORE LS BYTES OF RESULT
        LD    HL,(OP1)       GET TWO MS BYTES OP1
        LD    DE,(OP2)       GET TWO MS BYTES OP2
        SBC   HL,DE          SUBTRACT WITH BORROW
        LD    (OP1),HL       STORE MS BYTES OF RESULT
DONE    ~
        ↓
```

Notice that in the above subtract example the carry (borrow) had to be cleared by an OR A (or AND A) before the first subtract was done. The next subtract utilizes the carry (borrow) from the lower order. The 16-bit arithmetic operations resulted in a much more compact implementation of the problem. As the HL and other registers involved may be easily stored in the stack, use of the HL and other register pairs for n-precision arithmetic is very conveniently done.

The contents of IX and IY may be altered by an add from BC, DE, SP, or the index register itself. The obvious use for this is in *indexing through* tables, or other data structures in memory where data is located every nth byte in the table. If n is loaded into one of the register pairs, the index register may easily be altered to index to the next location. Let us see how this works. The following program searches a table of 128 entries for a given key value. Each entry is seven bytes long and the byte corresponding to the key is located at the third byte in the entry as shown in Fig. 11-1. If a *match* is found, the program exits to DONE with the match location in IX. If no match to the key is found, IX contains −1.

```
SRCH    LD    B,KEY          LOAD KEY VALUE
        LD    IX,TABLE+2     START OF TABLE + 2
        LD    C,128          INITIALIZE COUNT
        LD    DE,7           INCREMENT VALUE
LOOP    LD    A,(IX)         GET TABLE ENTRY
        CP    B              COMPARE TO KEY
        JP    Z,DONE         GO IF MATCH
        ADD   IX,DE          INDEX + 7
        DEC   C              DECREMENT COUNT
        JP    NZ,LOOP        GO IF NOT 128TH ENTRY
        LD    IX,-1          SET NOT FOUND FLAG
DONE    ~
        ↓
```

The 16-bit increments and decrements have been used in many examples above. They are generally straightforward except for one

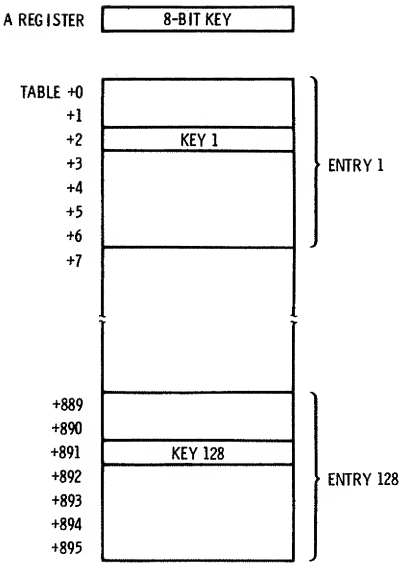


Fig. 11-1. Table search example.

caution. The *16-bit INCs or DECs do not affect any condition code flags.* One implication of this is that a register pair cannot easily be tested for a decrement of a count down to zero or a limit condition. If a register pair is to be used to hold a count, a 16-bit add or subtract may be used in place of the INC or DEC to increment or decrement the register pair, as these instructions *do* set the carry and, in some cases, the zero flag. Here is an example of how the ADD HL,SS instruction may be used to control the number of iterations through a LOOP.

```
        LD      DE,-1           LOAD -1 TO DE
        LD      IX,TABLE        START OF TABLE
        LD      HL,COUNT        LOAD COUNT OF N-1
LOOP    (
        (PROCESSING)
        )
        INC     IX              POINT TO NEXT BYTE OF TABLE
        ADD     HL,DE           DECREMENT COUNT BY 1
        JP      C,LOOP          CONTINUE
```

The count is initialized to *one less* than the number of iterations required. The last instruction of the routine tests the state of the carry. The next to last instruction effectively decrements the count in HL by one. No carry is produced (only if −1 is added to a 0 in

HL). Therefore, a condition of carry marks the termination of the loop after N iterations. The Z flag is not affected by the ADD HL,SS, but is affected by ADC and SBC.

GENERAL-PURPOSE ARITHMETIC INSTRUCTIONS

Two instructions in the general-purpose arithmetic group fall into a discussion of 8- and 16-bit arithmetic operations. The CPL and NEG instructions complement and negate the contents of the A register, respectively. The CPL forms the one's complement of the A register contents, while the NEG forms the two's complement of the A register contents. Both operations are convenient during normal processing in most programs.

DECIMAL ARITHMETIC OPERATONS

When an 8-bit add or subtract is performed, the arithmetic and logical unit in the Z-80 CPU performs a straight binary add or subtract. Some early computers performed bcd adds and subtracts

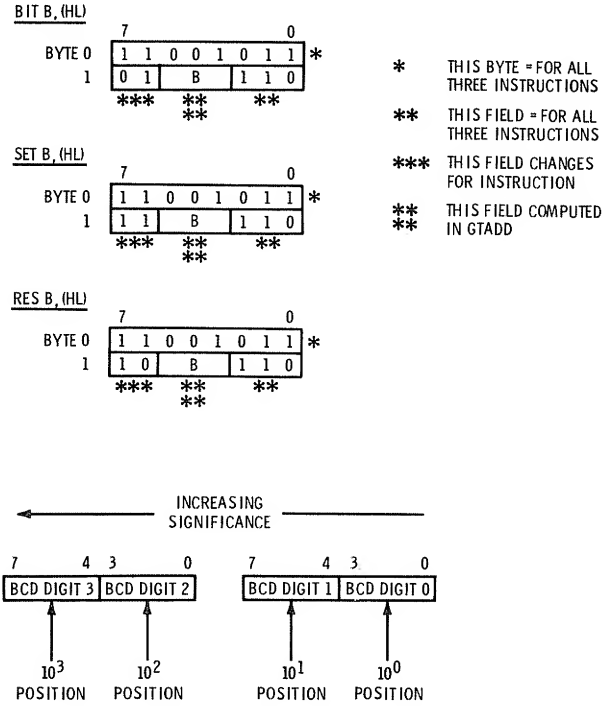


Fig. 11-2. Four-digit bcd representation.

rather than binary operations and data was retained in memory in bcd form. The Z-80 combines the simplicity of bcd representation with the efficiency of binary storage by the equivalent of bcd adds and subtracts. The binary add, or subtract, is first performed and then the DAA, or Decimal Adjust A instruction, is performed to adjust the binary result in the A register to bcd form. Multiple-precision bcd adds and subtracts may be performed as easily as multiple-precision binary operations. Suppose that two two-byte bcd operands are held in locations BCD1 and BCD2. Each operand consists of four bcd digits as shown in Fig. 11-2. The following code performs a bcd add of BCD2 to BCD1 with result stored in BCD1. The second example subtracts BCD2 from BCD1, stores results in BCD1.

```
BCDADD  LD    A,(BCD2+1)    GET LS BCD DIGITS OP2
        LD    B,A
        LD    A,(BCD1+1)    GET LS BCD DIGITS OP2
        ADD   A,B
        DAA                 BCD ADD
        LD    (BCD1+1),A    STORE LS RESULT
        LD    A,(BCD2)      GET MS BCD DIGITS OP2
        LD    B,A
        LD    A,(BCD1)      GET MS BCD DIGITS OP1
        ADC   A,B
        DAA                 BCD ADD
        LD    (BCD1),A      STORE MS RESULT
DONE    (
```

```
BCDSUB  LD    IX,BCD1+1     POINT TO LS OP1
        LD    IY,BCD2+1     POINT TO LS OP2
        LD    B,-2          LOOP COUNT
        OR    A,A           CLEAR CARRY
LOOP    LD    A,(IX)        GET BCD DIGITS OP1
        LD    B,(IY)        GET BCD DIGITS OP2
        SBC   A,B           OP1 - OP2
        DAA                 DECIMAL ADJUST
        LD    (IX),A        STORE RESULT
        DEC   IX            POINT TO HIGHER ORDER
        DEC   IY
        INC   B             BUMP LOOP COUNT
        JP    NZ,LOOP       CONTINUE
DONE    (
```

As the subtract example shows, the routines may be easily generalized to operate on $(N \times 2)$ bcd digits. The carry flag is always set by the CPU after the DAA operation to represent the carry from the *bcd operation* so that the carry (or borrow) is properly propagated to higher-order bcd operations.

CHAPTER 12

Shifting and Bit Manipulation— Rotate and Shift, Bit Set, Reset, and Test Groups

The instructions in this general category are basically concerned with shifting for arithmetic reasons or with manipulation of bits or fields of data. The Z-80 allows many combinations of shifts, supplementing the basic 8080 base containing four rotate A register instructions with logical and arithmetic shifts to CPU registers or memory. With the proper use of shifts, many arithmetic operations such as multiplication and division may be implemented in addition to manipulation of fields within words. The bit-oriented instructions permit testing and storage of data on a bit basis, either in CPU registers or memory.

LOGICAL SHIFTS

Logical shifts are perhaps the simplest shifts to understand. In a logical shift of eight bits, there is no consideration of the sign. The data is shifted right, or left, one bit at a time. The data is not "recirculated" to the opposite end of the register or memory location as it is shifted; bits that are shifted out of the register are lost except that they set the CY flag. Zeros fill vacated bit positions. There are two shift methods that perform a logical shift in the Z-80, one being the SRL or Shift Right Logical, and the other being the SLA or Shift Left Arithmetic. The latter, although designated "arithmetic," performs a classic logical shift. All shifts in the Z-80 operate on

eight bits of data; there are no 16-bit shifts except those implemented in software.

Logical shifts are usually implemented for one of two reasons:

1. To multiply and divide by powers of 2 or other factors.
2. To align fields within data bytes.

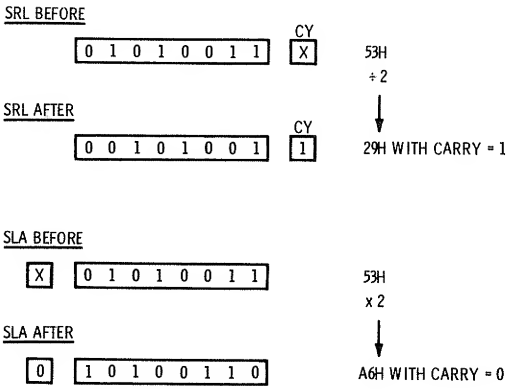


Fig. 12-1. Multiplication and division by shifting.

MULTIPLICATION AND DIVISION BY SHIFTING

Fig. 12-1 shows an SRL and SLA performed on the hexadecimal value 53H. The result after the SRL is 29H with the carry set. The result after the SLA is A6H with the carry reset. The effect of the SRL has been to divide by 2, while the SLA has multiplied by 2. For each bit position shifted right, the SRL divides by 2 so that n SRLs divide by 2^n. For each bit position shifted left, the SLA will multiply by 2; a shift of n bit positions divides by 2^n. As an example of this, consider the routine below. This routine finds the average of eight test scores where each test score represents a number from one to ten. Since the maximum total is 80, a single byte may be used for the total. The total is divided by eight by a shift right of three bit positions implemented by three SRLs. The average is a *truncated* average to the next lowest integer.

```
FNDAVE   LD    IY,TESTS-1   POINT TO TABLE OF TESTS
         LD    B,8          SET UP COUNT
         XOR   A            ZERO A FOR TOTALIZATION
LOOP     INC   IY           BUMP POINTER
         DEC   B            DECREMENT COUNT
         JP    M,JUMP1      GO IF 8 ITERATIONS
         ADD   A,(IY)       TOTALIZE
         JP    LOOP         CONTINUE
```

```
JUMP1   SRL   A          DIVIDE BY 2
        SRL   A          DIVIDE BY 4
        SRL   A          DIVIDE BY 8
DONE
          )
          \/

TESTS   DEFS  8          TABLE OF TEST SCORES
```

This program is implemented somewhat differently than previous loops. (Not as efficiently either!) Here the IY register and count are modified before the processing. A test is made of the minus state of the sign flag to terminate the loop. If the total number of test scores has not been processed, the next score is added and the loop continues. Note that the initial value in IY is equal to (TABLE − 1) but is equal to TABLE by the time the first score is retrieved.

By a combination of shifting and addition, multiplication or division by any number that can be written as a sum of powers of two is possible. A frequently seen use of this method is multiplication or division by 10 which can be factored into (8 + 2). The example below illustrates multiplication by 10 of an 8-bit number, assumed to be 25 or less to fit within an unsigned 8-bit byte.

```
MUL10   LD    A,NUMBER   GET MULTIPLICAND
        SLA   A          MULTIPLICAND × 2
        LD    B,A        SAVE
        SLA   A          MULTIPLICAND × 4
        SLA   A          MULTIPLICAND × 8
        ADD   A,B        MLCND*(8+2) = M*10
```

Logical shifts are commonly used to align data within fields, although in some cases a rotate and mask operation may be performed. The following routine is one method of converting two hexadecimal digits into their corresponding ASCII values of 0 − F. A logical shift is used on the first digit to align it (right justified) so that the ASCII conversion may be performed.

```
CVERT   LD    A,VALUE    GET TWO HEX DIGITS
        LD    B,A        SAVE
        SRL   A
        SRL   A
        SRL   A
        SRL   A          ALIGN FOR CONVERT
        ADD   A,3OH      CONVERT TO ASCII
        CP    A,3AH
        JP    M,OK1      GO IF NO CORRECTION
        ADD   A,7        CORRECT FOR A − F
OK1     LD    (BUF),A    STORE FOR OUTPUT
        LD    A,B        GET 2ND DIGIT
        AND   FH         MASK OUT 1ST DIGIT
```

```
        ADD   A,3OH        CONVERT TO ASCII
        CP    A,3AH
        JP    M,OK2        GO IF NO CORRECTION
        ADD   A,7          CORRECT FOR A - F
OK2     LD    (BUF+1),A    STORE FOR OUTPUT
DONE
```

A left logical shift may also be implemented by addition. The A register or HL register pair may be added to itself to shift data or simply to multiply by powers of two. The ADD A, instruction is one byte and takes 1 microsecond while a corresponding SLA is two bytes and takes 2 microseconds. Needless to say the ADD A should always be used. The 16-bit shifts of the HL register may be implemented by the ADD HL,HL instruction. The code below duplicates the multiply by ten above, but with 16 bits.

```
MUL10   LD    HL,(NUMBER)   GET MULTIPLICAND
        ADD   HL,HL         2 * MULTIPLICAND
        PUSH  HL
        POP   DE            TRANSFER TO DE
        ADD   HL,HL         4 * MULTIPLICAND
        ADD   HL,HL         8 * MULTIPLICAND
        ADD   HL,DE         10 * MULTIPLICAND
```

ROTATE-TYPE SHIFTS

The Z-80 has eight rotate-type shifts, some of which are redundant. The rotates basically shift the eight bits of the operand *and* a carry or eight bits of the operand alone. In the first case, the shift is really a 9-bit shift if the carry is considered an extension of the register or memory location. Both the rotates with the carry and the rotates without the carry have their uses. All rotates preserve all eight or nine bits of the data and shift a CPU register or memory operand

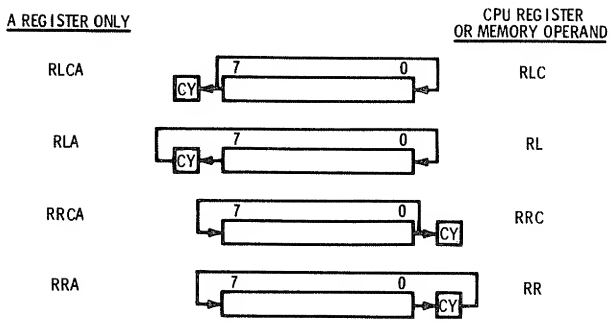


Fig. 12-2. Shift actions.

1-bit position to the right or left. For those shifts that do not shift through the carry, the carry is set to the value of the bit shifted out of the register and around. Fig. 12-2 recaps the shift actions.

Rotate shifts are used to align fields within bytes for access and storage and to enable multiple-register shifts. Data in a CPU register or memory location may be tested by a rotate shift, but not necessarily destroyed.

Discounting the fact that we can obtain the parity of an 8-bit operand very easily by performing an AND A or OR A, let us illustrate how a rotate may be used to compute the parity of a memory operand.

```
PARITY   XOR   A          CLEAR PARITY AND C
         LD    B,8        INITIALIZE COUNT
         LD    HL,MEMOP   MEMORY OPERAND ADDRESS
LOOP     RLC   (HL)       SHIFT OUT BIT TO CY
         JR    NC,JUMP1   GO IF NOT A ONE BIT
         XOR   1          FLIP PARITY INDICATOR
JUMP1    DEC   B          DECREMENT COUNT
         JR    NZ,LOOP    GO IF NOT 8 BITS
DONE     )                A REGISTER NOW 0 IF EVEN
         (                  # OF 1 BITS, 1 IF NOT
         v
```

The A register and carry are first cleared by the XOR. Now the RL (HL) instruction is executed eight times. At the end of eight times, the contents of MEMOP are identical to the contents before the routine was entered. The A register lsb was set or *toggled* each time a one bit was shifted around to bit 0, so that after eight shifts the A register bit 0 holds a one if the total number of one bits was odd, or a 0 if the total number of one bits was even.

Another example of the use of the rotate is shown next. Here, the rotate is used to convert an 8-bit binary operand to binary-ASCII

```
BXASB    EQU   $          BINARY-TO-ASCII-BINARY
         LD    A,(BYTE)   GET BYTE TO CONVERT
         LD    C,8        SET # COUNT
         LD    IX,BUF+7   BUFFER, LAST BYTE
LOOP     LD    B,3OH      ASCII 0
         RRCA             SHIFT OUT BIT
         JR    NC,JUMP1   GO IF BIT = 0
         INC   B          CHANGE TO ASCII 1
JUMP1    LD    (IX),B     STORE ASCII CHARACTER
         DEC   IX         POINT TO HIGHER-ORDER SLOT
         DEC   C          DECREMENT COUNT
         JR    NZ,LOOP    GO IF NOT 8 TESTS
DONE     )
         (
         v
```

digits. As there will be eight ASCII characters representing 0 (ASCII 3OH) or 1 (ASCII 31), a buffer of eight bytes is allocated to hold the results of the conversion. The bits in the operand to be converted are tested one at a time by shifting the A register right in a rotate shift. The rotate could be either an RRCA rotate or an RRC A rotate. Both perform the same action (the RRCA is compatible with the 8080 rotate of this kind). Since the RRCA takes one byte and 1 microsecond and the RRC A takes two bytes and 2 microseconds, the RRCA was chosen.

If the value at BYTE was 10111001_2, the character stored at BUF through BUF+7 for this routine would be 31, 30, 31, 31, 31, 30, 30, 31, all hexadecimal.

The rotate shifts may be used in conjunction with logical shifts to facilitate multiple-precision shifts. Suppose that we wish (for some unfathomable reason) to shift a 3-byte operand located at UDGE two bit positions to the left in a logical shift. The rotate may be used to propagate any carry along the 3-byte chain as follows:

```
SLUDGE  EQU   $           SHIFT LEFT UDGE
        LD    IX, UDGE+2
        SLA   (IX)        0 TO BIT 0, BIT 7 TO C
        RL    (IX-1)      C TO BIT 0, BIT 7 TO C
        RL    (IX-2)      C TO BIT 0, BIT 7 TO C
        SLA   (IX)        0 TO BIT 0, BIT 7 TO C
        RL    (IX-1)      C TO BIT 0, BIT 7 TO C
        RL    (IX-2)      C TO BIT 0, BIT 7 TO C
```

Another variation of this implementation uses the HL register.

```
SLUDGE  LD    HL,(UDGE+1)   TWO LS BYTES
        LD    IY, UDGE
        ADD   HL,HL         SHIFT LEFT 1 TO C
        RL    IY            C TO MS BYTE
        ADD   HL,HL         SHIFT LEFT 1 TO C
        RL    IY            C TO MS BYTE
        LD    (UDGE+1),HL   STORE LS BYTE
```

ARITHMETIC SHIFTS

We have already covered the shift left arithmetic in previous examples. Various manufacturers implement arithmetic left shifts in one of two ways. Many equate a left arithmetic shift to a left logical and leave it go at that—the sign is simply shifted out on the first shift. Other manufacturers retain the sign on a left shift. The bit in bit position six is shifted out into the carry and bit seven remains.

In any event, the Shift Right Arithmetic is unambiguous. The sign

bit position, bit seven, *is* retained, *and* extended into bit position six. Fig. 12-3 shows a large negative number reduced by shifting. As a right arithmetic shift is performed, the number in the register or memory operand is truncated, that is, bits that are shifted off the right are lost. In fact, the instantaneous value of the carry after a right shift represents the remainder of a divide-by-two operation. If the fractional portion of the arithmetic portion of the shift is to be retained, the bits shifted off must be saved in another register.

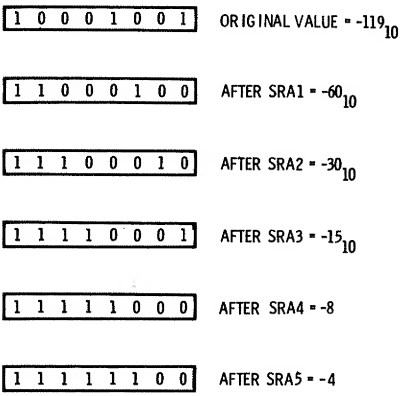


Fig. 12-3. SRA action.

The following routine performs an arithmetic right shift and saves the fractional part of the number in the B register. If the binary point is considered to be between the two registers, then the bit positions of the fractional portion represent 1/2, 1/4, 1/8, etc.

```
ARSSVF   LD    A,(NUMBER)   GET NUMBER
         LD    B,0          CLEAR FRACTION
         SRA   A            N/2
         RR    B            SAVE 1/8
         SRA   A            N/4
         RR    B            SAVE 1/4
         SRA   A            N/8
         RR    B            SAVE 1/2
DONE     (
```

The effect of an arithmetic right shift on a positive number is to truncate the number and retain the quotient of the divide while ignoring the remainder. An example of this is the divide $1011_2/4$ implemented by a 2-bit arithmetic right shift (this is not a value judgement). The number after the shift is 2; the remainder of 3/4 has been lost. The effect of an arithmetic right shift on a negative number is to *round up* if the fractional part is ignored when the

number is converted to positive form. The divide $11110101_2/4$ is 11111101_2 with a remainder of 01000000_2 if the fractional remainder is saved. 11111101_2 is -3; the quotient has been rounded up to the nearest integer.

Since the arithmetic right shift extends the sign bit position, it normally should not be used to align data in fields. Erroneous one bits will be extended to the right if a negative number is arithmetically shifted prior to masking and testing.

THE 4-BIT BCD SHIFT

All of the above shifts shift one bit position to the right or left per shift instruction. The RLD and RRD instructions perform a *4-bit* shift to the left and right respectively, working with the A register and the location referenced by the HL register. The instructions are recapped in Fig. 12-4. Although this shift may be used conveniently for any processing that deals with 4-bit fields, it is very convenient for processing bcd data. Each 4-bit shift brings in a new bcd digit. Let us see an example of the way the shifts may be used to manipulate bcd data.

The program below converts ASCII characters, assumed to be the digits $0 - 9$, into bcd digits. Ten characters, representing a 10-digit bcd value of $0000000000_{10} - 9999999999_{10}$ are stored in INBUF through INBUF+9. The conversion will put the 5-byte *packed* bcd result in INBUF through INBUF+4, two bcd digits per byte, in the same order. See Fig. 12-5.

```
BCDAXB   EQU   $          BCD ASCII TO BCD
         LD    IX,INBUF   POINT TO BUFFER
         LD    HL,INBUF   POINT TO BUFFER
         LD    B,5        SET UP COUNT
LOOP     LD    A,(IX)     GET CHARACTER
         SUB   A,3OH      CONVERT TO BCD 0 - 9
         RLD              ROTATE TO (HL)
         LD    A,(IX+1)   GET NEXT CHARACTER
         SUB   A,3OH      CONVERT TO BCD 0 - 9
         RLD              ROTATE TO HL
         INC   IX         BUMP CURRENT CHARACTER
         INC   IX         POINTER
         INC   HL         BUMP STORAGE POINTER
         DEC   B          DECREMENT COUNTER
         JR    NZ,LOOP    GO IF NOT DONE
DONE     (
```

In the above program, the ASCII characters were accessed two per iteration through the loop. Each was converted to bcd by sub-

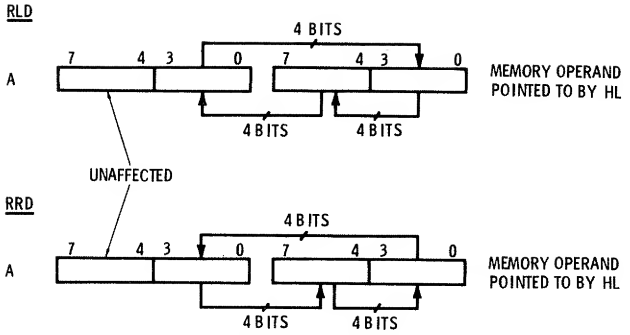


Fig. 12-4. RLD, RRD action.

traction of 3OH and the resulting bcd digits were rotated into the location pointed to by HL. At the end of the loop, the HL pointer was incremented by one and the IX by two; HL is the pointer to the next storage location, while IX points to the current ASCII character to be processed. The loop has five iterations to process five sets of two characters at a time.

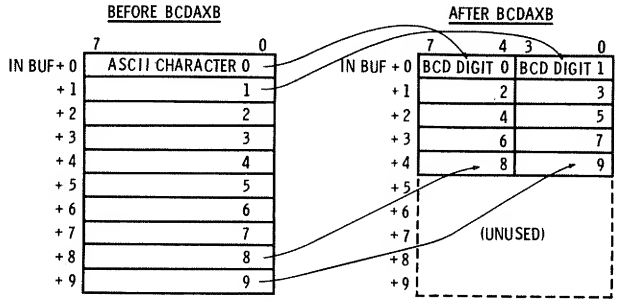


Fig. 12-5. ASCII to bcd conversion.

As a further example of the use of the 4-bit rotates, let us convert the opposite way, going from packed bcd digits in INBUF through INBUF+4 into ASCII characters representing the digits 0 through 9. Here, we must go backwards through the buffer as we are filling two bytes of ASCII data with each packed byte of two bcd digits. IX will hold the address of the next location to be used for storage in the buffer, while HL will point to the current pair of bcd digits. As two bcd digits will be processed at a time, five iterations will take place. The first two bcd digits are picked up from INBUF+4 and stored in INBUF+9 and INBUF+8. The conversion will continue until the digits at INBUF are accessed and the ASCII characters that result are stored at INBUF+1 and INBUF as shown in Fig. 12-6.

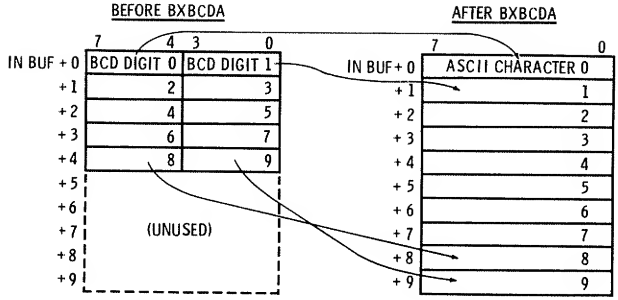


Fig. 12-6. Bcd to ASCII conversion.

```
BXBCDA   EQU  $          BCD TO BCD ASCII
         LD   IX,INBUF+9 PNT TO LST BYTE OF BUFFER
         LD   HL,INBUF+4 POINT TO FIRST TWO DIGITS
         LD   B,5        SET UP COUNT
LOOP     XOR  A          CLEAR A
         RRD              GET MS DIGIT TO A
         ADD  A,3OH      CONVERT TO ASCII
         LD   (IX),A     STORE IN BUFFER
         XOR  A          CLEAR A
         RRD              GET LS DIGIT TO A
         ADD  A,3OH      CONVERT TO ASCII
         LD   (IX-1),A   STORE IN BUFFER
         DEC  IX         POINT TO NEXT
         DEC  IX         STORAGE
         DEC  HL         POINT TO NEXT 2 DIGITS
         DEC  B          DECREMENT COUNTER
         JR   NZ,LOOP    GO IF NOT DONE
DONE     ~
```

The only subtlety in the program above is that the A register *must be cleared* before each bcd digit is shifted in, as the shift does not affect bits 7 – 4 of the A register.

BIT SET, RESET, AND TEST GROUP

The instructions in this group enable any of the eight bits in a CPU register or memory operand to be tested, set, or reset. Register indirect or indexed addressing is permitted for addressing operands in memory. The instructions in the group are rather powerful as they enable fast and efficient bit manipulation. Bits within bytes may be operated on by combinations of load and masking operations, rather than the bit instructions, but the resultant code is at least three instructions as shown here:

```
BITEST  EQU   $          TEST BIT
        LD    HL,BYTE    POINT TO BYTE
        LD    A,(HL)     GET BYTE
        AND   A,VALUE    MASK OUT BIT
VALUE   EQU   1          (OR 2, 4, 8, 16, 32, 64, 128)
```

```
BITSET  EQU   $          SET BIT
        LD    HL,BYTE    POINT TO BYTE
        LD    A,(HL)     GET BYTE
        OR    A,VALUE    SET BIT
        LD    (HL),A     STORE BYTE
```

```
BITRST  EQU   $          RESET BIT
        LD    HL,BYTE    POINT TO BYTE
        LD    A,(HL)     GET BYTE
        AND   A,NOTVAL   RESET BIT
        LD    (HL),A     STORE BYTE
NOTVAL  EQU   FEH        (OR FD, FB, F7, EF, DF, BF, 7F)
```

Each of the three routines above can be replaced with one equivalent bit test, set, or reset instruction. To test bit 7 in an 8-bit byte in memory, the following code is used:

```
        LD    HL,BYTE    POINT TO BYTE
        BIT   7,(HL)     TEST MS BIT
        JR    Z,ZERO     GO IF BIT IS A ZERO
ONE                      BIT IS A ONE
```

To set a bit in a memory operand,

```
        LD    IX,BYTE    POINT TO BYTE
        SET   5,(IX)     SET BIT 5
```

To reset a bit in a memory operand,

```
        LD    IY,BYTE    POINT TO BYTE
        RES   1,(IY)     RESET BIT 1
```

Of course, any of the three bit instructions may also be used to test, set, or reset any bit in a CPU register:

```
        BIT   7,B        TEST MS BIT OF B REGISTER
```

As an example of how the bit processing instructions may be used, consider the following example. A 256 column by 256 line video display is being used in a Z-80 microcomputer system as shown in Fig. 12-7. Each *pixel*, or picture element, is represented by one bit in

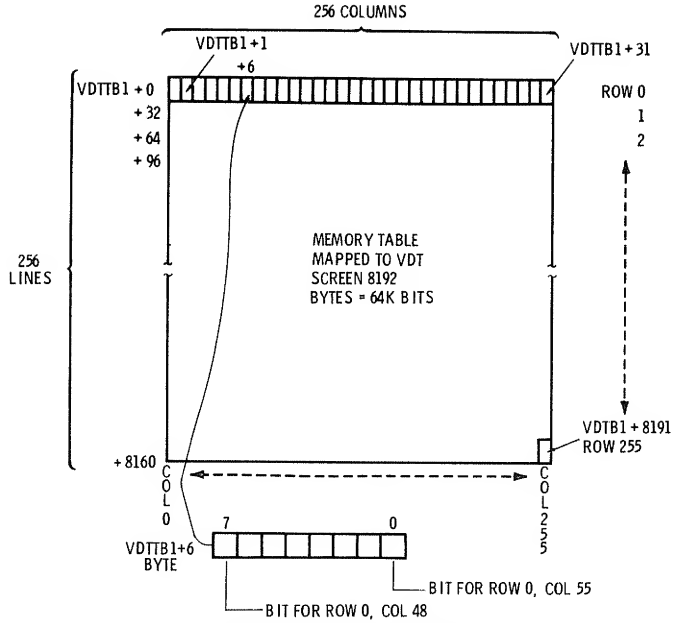


Fig. 12-7. VDT bit map for 64K pixels.

memory and can be on (white) or off (black). The first line is represented by the data at VDTTB1 through VDTTB1+31, each byte holding eight pixels worth of data. The entire 8K bytes (64K bits) is output to the VDT by direct memory access of the VDTTB1 data by the electronics in the VDT controller. The display program will continually update the display buffer VDTTB1 to change the display. For display of plots and other data, it is convenient to address the pixels by row, column representation. The pixel in the upper left-hand corner is 0,0 (row 0, column 0), the pixel in the upper right-hand corner is 0,255, the pixel in the lower left is 255,0, and the pixel in the lower right corner is 255,255. The problem is to convert a pixel address from row, column representation to the actual bit address and access the bit in VDTTB1 to test the current value, set the bit, or reset the bit to change the display. The following examples are three portions of the program. Data is *passed to* the programs in the A register and B register, A representing the row and B the column. The set and reset programs simply set or reset the required pixel. The test *passes back* an *argument* representing the current value of the pixel in the Z flag, Z=0 for on (white), and 1 for off (black).

```
;  THIS ROUTINE TESTS THE VALUE OF A PIXEL
TESTPX      LD      C,46H           LOAD SKELETON (BIT)
JUMP        CALL    GTADD           GET ADD IN HL, BIT # IN A
```

```
                OR      A,C             MERGE BIT #
                LD      (MODIFY),A      STORE BIT INSTRUCTION
INSTRU          BIT     0,(HL)          TEST, SET, OR RESET BIT
                RTN                     RETURN
MODIFY          EQU     INSTR+1         SEC BYTE OF BIT INSTR
;  THIS ROUTINE SETS THE VALUE OF A PIXEL TO 1
SETPX           LD      C,C6H           LOAD SKELETON (SET)
                JMP     JUMP
;  THIS ROUTINE RESETS THE VALUE OF A PIXEL TO 0
RESTPX          LD      C,86H           LOAD SKELETON (RES)
                JMP     JUMP
;  THIS ROUTINE PUTS ADDRESS OF BYTE CONTAINING
;  PIXEL IN HL, BIT # IN A ALIGNED IN BITS 5 - 3
GTADD           PUSH    BC              SAVE B,C IN STACK
                SRL     A               TRUNCATE 3 LSBS OF B
                RR      B               ALIGN 3 ADDRESS BITS
                SRL     A
                RR      B
                SRL     A
                RR      B
                LD      L,B             DISPLACEMENT NOW IN HL
                LD      H,A
                ADD     HL,VDTTB1       VDT TABLE ADDRESS
                POP     BC              RESTORE B,C
                LD      A,B             GET LOWER-ORDER BITS
                CPL
                AND     7               GET BIT ADDRESS
                SLA     A               ALIGN BIT ADDRESS
                SLA     A               FOR BIT INSTRUCTION
                SLA     A
                RET                     RETURN
```

These routines are probably more sophisticated than any we have considered thus far. In the first place, they are true subroutines, callable by a CALL instruction. The return is made with the RTN instruction. There are three *entry* points in the subroutine TESTPX which tests the value in a pixel, SETPX, which sets a 1 into the pixel, and RESTPX, which resets a pixel to 0. Depending on the function to be performed, they are callable by something similar to:

```
                LD      A,ROW           GET CURRENT ROW
                LD      B,COL           GET CURRENT COLUMN
                CALL    SETPX           SET PIXEL TO ON
                 (                      RETURN HERE
                 ↓
```

Subroutine TESTPX (which also encompasses the SETPX and RESTPX functions) calls yet another subroutine GTADD. Subroutine GTADD converts a row, column address to a byte address in

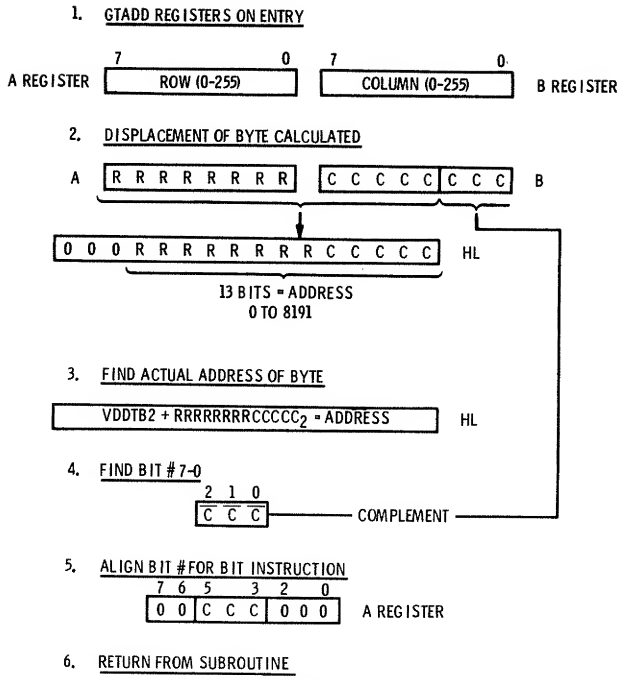


Fig. 12-8. Subroutine GTADD action.

HL and a bit address in A. Let us investigate GTADD in detail first. The parameters on input to GTADD are as shown in Fig. 12-8, the row # 0 to 255 in A and the column # 0 to 255 in B. The byte *displacement* of the byte containing the pixel bit is given by (ROW $\times$ 32) + (COLUMN/8). The bit address 0 – 7 (most significant bit to least significant bit) is given by the remainder of COLUMN/8. GTADD calculates the byte displacement by shifting A and B and then calculates the actual address within the table by adding the displacement to the address of the start of the table. The byte address is then stored in HL. The bit address of the pixel is found by masking the three least significant bits of the column address in B. Since these are the inverse of the Z-80 bit position numbers, they are complemented before the mask to set the corresponding Z-80 bit position numbers. After the bit position number is found, it is aligned to bits 5 – 3 of the A register for reasons which shall become obvious. The entire action of subroutine GTADD is shown in Fig. 12-8.

To test the value of a pixel, subroutine TESTPX is called. A and B contain the row and column of the pixel, respectively. The first action taken is to load the C register with a value corresponding to the second byte of a BIT instruction. Value 46H represents the second

byte of BIT 0,HL. Subroutine GTADD is then called to compute the byte and bit address of the pixel within VDTTB1. Aligned bit number in A on return is ORed with BIT 0,HL code in C to produce the BIT B,HL form of the instruction. This is stored in the second byte of the BIT instruction (location MODIFY). The BIT 0, (HL), now a BIT B,(HL) instruction, is then executed setting the Z flag on the results of the bit test. The Z flag is still valid when the RTN instruction is executed to return to the main calling program.

SETPX and RESTPX operate in similar fashion, except that the instructions loaded in C are the second bytes of SET 0,(HL) and RES 0,(HL). The B field is merged on the OR after the return from GTADD. The three instructions executed at INSTRU for the three entry points are shown in Fig. 12-9.

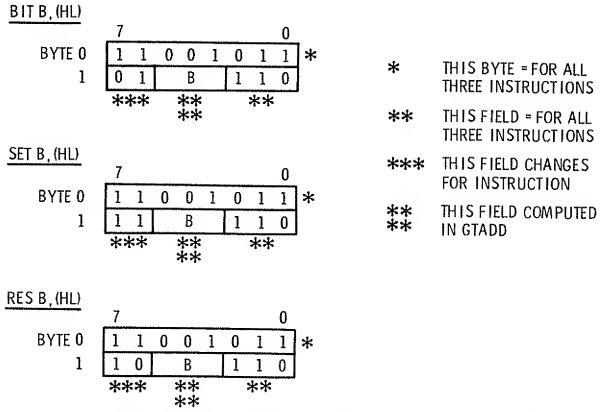


Fig. 12-9. Instruction modification for VDT bit routine.

The previous program utilizes many of the shifting, bit manipulation, indexing, and logical functions discussed in this and previous chapters. It also introduces several new concepts. One of the most important is that an instruction is simply data that can be modified as any other data value. The instruction at INSTRU was *dynamically* changed rather than changed at assembly time to reflect the function performed. The above implementation also illustrates the use of subroutines and subroutine calls. This subject will be covered in more detail in Chapter 14.

SOFTWARE MULTIPLICATION AND DIVISION

The subject of software multiply and divide is discussed in this chapter because the implementation of these functions, in the general case, is primarily shift and bit test operations. There are several kinds of implementations possible for multiply and divide opera-

tions, but most are inefficient. The multiply by shifting and adding powers of two works well for decimal and other multipliers. The inverse of this, shifting and subtracting powers of two, also will work well for division of small divisors. Other methods for multiplication and division include successive addition (multiplication) and subtraction (division) and a variation of this based on table "lookup" of powers of ten. All of these implementations, however, suffer in the general case from large execution times. As an example of the execution times involved, consider the following program which divides by successive subtraction of the 8-bit unsigned divisor from a 16-bit unsigned dividend.

```
DIVIDE  LD    HL,(DIVDND)   16-BIT DIVIDEND
        LD    A,DVISOR      8-BIT DIVISOR
        NEG                 NEGATE DIVISOR
        LD    C,A           - DIVISOR TO BC
        LD    B,FFH
        LD    DE,0          CLEAR QUOTIENT
LOOP    ADD   HL,BC         SUBTRACT DIVISOR
        JR    NC,DONE       GO IF DONE
        INC   DE            BUMP QUOTIENT
        JP    LOOP          CONTINUE
DONE    (
        v
```

The divisor is effectively subtracted from the dividend until the *residue* goes below 0. For each successful subtract, the quotient is incremented by one. The best case execution of this program is about 25 microseconds. The worst case time is about 1/2 second! With an average time of about a millisecond, the program is far too slow for software that requires many divide operations.

The generic form of most multiply routines emulate a paper and pencil multiplication exercise. The digits of the multiplier are examined one at a time, multiplied against the multiplicand, the product added to a partial product, and a shift made to the next digit

```
MULT    LD    L,0           CLEAR L
        LD    H,A           MULTIPLIER TO H
        LD    C,B           MULTIPLICAND TO C
        LD    B,0           0, MULTIPLICAND TO B,C
        LD    A,8           ITERATION COUNT
LOOP    ADD   HL,HL         SHIFT LEFT ONE
        JR    NC,JUMP1      GO IF NO CARRY
        ADD   HL,BC         ADD MULTIPLICAND TO PARTIAL PROD
JUMP1   DEC   A             DECREMENT ITERATION COUNT
        JR    NZ,LOOP       GO IF NOT 8 ITERATIONS
DONE    (
        v
```

position. Let us see how this works in a simple unsigned *multiply* of two 8-bit operands. The multiplier is passed in the A register and the multiplicand in the B register.

The register arrangement at the start of the multiply is shown in Fig. 12-10. The multiplier is shifted out one bit at a time. If the multiplier bit is a one (carry set), the multiplicand is added to the partial product; if a zero, no add is made. After eight iterations, the multiplier has been shifted completely out of H and HL holds the 16-bit product. The worst case time for this multiply is under 100 microseconds, a factor of 10 better than for a successive addition method. The same general implementation may be carried out for multiplies of greater widths although a product exceeding 16 bits will require additional code for shifting more than one register pair.

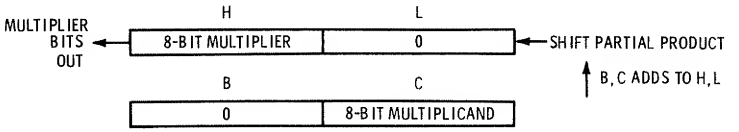


Fig. 12-10. An 8-bit multiply register arrangement.

The general form of division of the Z-80 and similar microprocessors is also related to the paper and pencil method. In this case, the *restoring* division of manual methods is used. The *residue* or partial dividend is examined to see if the divisor "will go" into it (is less than or equal to the dividend). If the subtract can be made, a one bit is put into the quotient and a shift is made for the next divide. If the subtract cannot be made, the value of the residue is restored by adding back the dividend and a zero bit is put into the quotient. The following routine divides a 16-bit unsigned dividend in HL by an 8-bit unsigned divisor in the B register to yield an 8-bit quotient and 8-bit remainder.

```
DVIDE8    LD      C,0       NOW - DIVISOR IN BC
          LD      D,8       ITERATION COUNT
LOOP      ADD     HL,HL     SHIFT LEFT ONE
          XOR     A         CLEAR C
          SBC     HL,BC
          INC     HL        SET Q=1
          JP      P,JUMP1   GO IF POSITIVE RESULT
          ADD     HL,BC     RESTORE
JUMP 1    RES     0,L       RESET Q BIT TO 0
          DEC     D         DECREMENT COUNT
          JR      NZ,LOOP   CONTINUE IF NOT 8
DONE      ~
```

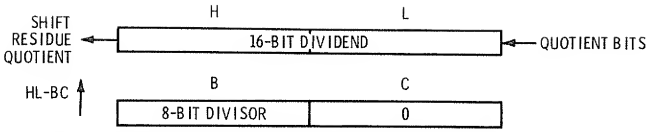


Fig. 12-11. A 16-bit by 8-bit divide register arrangement.

The register arrangement at the start of the divide is shown in Fig. 12-11. The divisor is subtracted from the residue in HL after a left shift of the residue (0 is in the msb of the dividend for the first shift). The quotient bit is preset to a 1 in the lower end of the L register. If the subtract will not "go," a restore is done (ADD HL,BC) and the quotient bit reset to 0. At the end of the divide, the residue or remainder is in H and an 8-bit quotient has been shifted into L. Overflow is possible if the quotient cannot be resolved in eight bits.

The preceding is a brief introduction to implementations of unsigned "multiplies" and "divides." In many software projects such simple operations will suffice; in other systems, more elaborate arithmetic operations such as floating-point implementations will be required.

CHAPTER 13

List and Table Operations— Search Group

This chapter discusses three kinds of *data structures*, strings, tables and lists, and the techniques used to access them. The search group of Z-80 instructions are specifically implemented to search through tables and strings of data, and they are described in detailed examples in the discussion.

DATA STRINGS

Data strings are sequences of data, and the common usage refers to character data. The assembler pseudo-operation TXT generates a string of ASCII characters as in the following example:

```
MESG1 TXT $THIS IS A CHARACTER STRING$
```

String operations are important in text processing and compiler operation, and some higher-level languages have been implemented specifically to deal with string manipulation.

The Z-80 has the capability to search a sequence of data bytes for a given byte. The search may be made for character or other data as the implementation is only concerned with finding a data byte that matches a search key. The implementation of the search instructions is very similar to the implementation of other Z-80 block-related instructions. An 8-bit search key is loaded into the A register. The HL register pair is loaded with the starting address of the data string, and the BC register pair is loaded with the number of bytes to be searched. If the CPI instruction is used, the search will

be done "semi-automatically," that is, one instruction will be executed. The byte pointed to by HL will be accessed and compared to the contents of the A register. The contents of HL will be incremented and the contents of BC will be decremented. If BC $\neq 0$ after the decrement, the P/V flag will be set. The S, Z, and H (half-carry) will be set if the comparison result is negative, equal, or produces a half-carry, respectively. The next instruction in sequence will then be executed. Examples of CPI use will be presented under table operations in this chapter.

If the instruction is a CPIR instruction, all of the setup and execution details are the same, except that the CPIR will continue execution until either a match is found with the search key in the A register or until the byte count has been decremented down to zero (BC = 0). Testing of the P/V and Z flags will indicate the terminating condition.

As an example of CPIR use, consider the following example. A string of 64 characters, starting at STRING, is to be searched for the character "$."

```
SRCHD   LD    A,24H       DOLLAR SIGN
        LD    HL,STRING   ADDRESS OF FIRST CHAR
        LD    BC,64       64 BYTES MAXIMUM
        CPIR              SEARCH STRING FOR $
        JP    Z,FOUND     GO IF CHARACTER FOUND
        ⌒                 NOT FOUND
        ↓
FOUND   DEC   HL          POINT TO $
        LD    (PNTR),HL   SAVE POINTER TO $
```

The A register is loaded with the hexadecimal equivalent of an ASCII dollar sign. Register pair HL is initialized with the address of the start of the character string, STRING. Since 64 characters are to be searched, a byte count of 64 is loaded into BC. The CPIR sequences through the 64-character string. If a dollar sign is found, CPIR is exited to the next instruction with the Z flag set. If the dollar sign was at the last character, *both* the Z flag and the P/V flag will indicate terminating conditions; the Z flag will be set indicating the character was found and the P/V flag will be reset indicating that the byte count was decremented down to zero. It is necessary to test the Z flag first, therefore, to see if the character was found before testing the P/V for end of string. If the character is found, the HL register points to the byte *after* the character, so the pointer must be adjusted by one to point to the location of the dollar sign. The terminating conditions for a typical successful search for this example are given in Fig. 13-1.

As described in a previous chapter, the CPDR works in similar fashion to the CPIR, except that it *decrements* from the end of the

list. The contents of HL is decremented after each iteration of the instruction; the contents of BC is, of course, decremented as in the CPIR. The initialization for a 64-character string for use with a CPDR is as follows.

```
SRCHD   LD      A,24H           DOLLAR SIGN
        LD      HL,STRING+63    ADDRESS OF LAST CHAR
        LD      BC,64           64 BYTES MAXIMUM
        CPDR                    SEARCH FOR $ BACKWARDS
```

CPIR and CPDR work extremely efficiently for one-character keys. The time per iteration if the key is not found is 5.25 microseconds, making the average search time through a 64-character string $(64/2) \times 5.25 = 168$ microseconds. The corresponding code for an access, compare, conditional test, adjust of byte count and pointer, and conditional test would be about double the CPIR or CPDR and occupy a great deal more memory.

If a key of more than one character is required, the compare string instructions may also be used, but the comparison process will not be quite as efficient as a search for a single byte. If the first character search is made on a frequently occurring character (such as

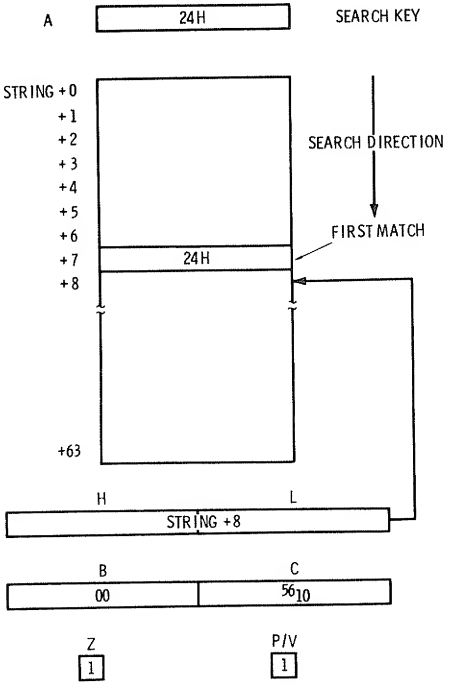


Fig. 13-1. String search terminating conditions.

an "e" in an alphanumeric string), then a significant amount of time will be wasted in comparing the remainder of the string to the search string; if the first character of the search string occurs less frequently (such as a "q"), then the number of "failing comparisons" will be fewer and the overall search more efficient. One general way to search for a given string of more than one character is given in the next sample instructions. Here, a search is made for a two-character string. The first level of the search is performed for the first letter of the key string. When the first letter is found in the string to be searched, the next character is compared to the second key character. If a match is made, the routine exits; but if a match is not made, the routine restarts from the point at which the first character was found. Here, the search is through a string of 64 characters. The search is over if no match has been found by the 63rd character.

```
BIGSRC   LD     HL,STRING      START OF STRING
         LD     BC,63          63 BYTES MAXIMUM
LOOP     LD     A,(CHAR+0)     FIRST CHARACTER OF KEY
         CPIR                  SEARCH FOR 1ST CHAR
         JP     NZ,NFND        GO IF NOT FOUND
SECLVL   LD     D,(HL)         GET SECOND CHAR
         LD     A,(CHAR+1)     GET SECOND CHAR OF KEY
         CP     A,D            COMPARE
         JP     NZ,LOOP        GO IF NOT FOUND
FOUND    )                     HL POINTS TO MATCH + 1
         \/
```

If the first key character has a match, then the second level comparison at SECLVL is entered. HL at this point points to the match plus one so that the next character can be picked up directly. A is then loaded with the second key character and a comparison made. If there is a compare, FOUND is executed; if there is no compare, LOOP is reentered. The BC and HL registers are already properly set for the next CPIR comparison in the no match case!

A variation of the above technique can be used to search for a given length key or a *hashing* comparison may be made for the second level compare. In the latter case, a one-for-one compare of each of the remaining bytes in the key and string is not made. The remaining bytes of the string are used to compute a *hash value* which is then compared with a precomputed key hash value. If the two hash values are equal, then a direct compare is made. If the two hash values are not equal, then the search continues. The hash algorithm may be any scheme that produces a relatively unique value. If the hash algorithm adds the remaining bytes together, for example, a relatively unique hash value is found as shown in Table 13-1. Here, about 1 in 30 five-character strings will produce the same hash value, as is the case for "BROWN" and "MAUVE."

Table 13-1. Five-Character String Comparison

Character String	ASCII					Total of Bytes
WHITE	57H,	48,	49,	54,	45	181H
GREEN	47,	52,	45,	45,	4E	171H
BROWN	42,	52,	4F,	57,	4E	17EH
MAUVE	4D,	41,	55,	56,	45	17EH

An implementation of a search based on this simple hash is shown in the following code.

```
HASHSR  LD      IX,CHAR         START OF 5-CHAR KEY
        LD      A,(IX+1)        LOAD KEY CHARACTERS
        ADD     A,(IX+2)
        ADD     A,(IX+3)
        ADD     A,(IX+4)
        LD      D,A             SAVE IN D
        LD      HL,STRING       START OF STRING
        LD      BC,60           60 BYTES = DONE
LOOP    LD      A,(CHAR+0)      FIRST CHARACTER
        CPIR                    SEARCH FOR 1ST CHAR
        JP      NZ,NFND         GO IF NOT FOUND
SECLVL  PUSH    HL              SAVE POINTER
        POP     IX              POINT TO 2ND CHAR
        LD      A,(IX)
        ADD     A,(IX+1)
        ADD     A,(IX+2)
        ADD     A,(IX+3)        COMPUTE HASH
        CP      D               COMPARE TO KEY HASH
        JP      NZ,LOOP         GO IF NO MATCH
MAYBE   LD      IY,CHAR+1       POINT TO 2ND CHAR OF KEY
        LD      A,(IX)
        CP      A,(IY)
        JP      NZ,LOOP         GO IF NO MATCH 2ND CHAR
        LD      A,(IX+1)
        CP      A,(IY+1)
        JP      NZ,LOOP         GO IF NO MATCH 3RD CHAR
        LD      A,(IX+2)
        CP      A,(IY+2)
        JP      NZ,LOOP         GO IF NO MATCH 4TH CHAR
        LD      A,(IX+3)
        CP      A,(IY+3)
        JP      NZ,LOOP         GO IF NO MATCH 5TH CHAR
FOUND   (                       FOUND!
```

If a match is found on the first character, SECLVL is entered. A hash is then computed for the remaining four characters of the string and compared to the precomputed key hash in D. If there is

no match, LOOP is reentered. If there is a match, MAYBE is entered where a direct one-for-one comparison is made culminating in FOUND, if all of the remaining characters match. The portion of code from MAYBE to FOUND could have been shortened considerably by subroutine or loop use, but the code was expanded for clarity.

TABLE OPERATIONS

Strings are a subset of tables, as they are contiguous lists of items. Each entry in the string consists of one ASCII character. A table is

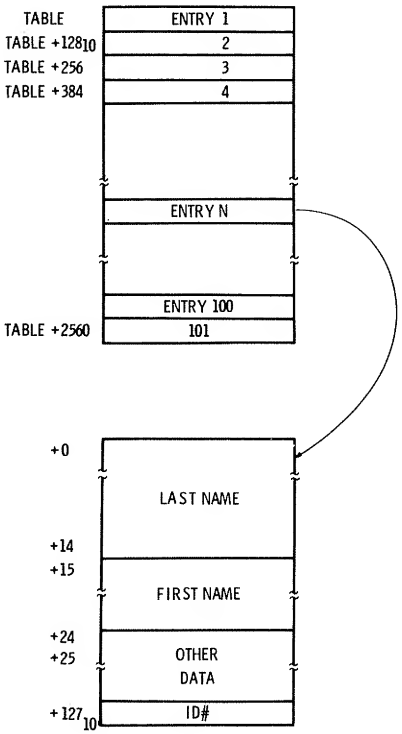


Fig. 13-2. Employee table format.

made up of a number of entries that may be any number of bits long and may contain subgroupings of data. Within the table, entries may be ordered, in random order, or indexed by some external key. Ordering may be done on any subgroup within the table entry. Let us use a table of names and addresses to illustrate the use of an indexed table in the Z-80. LASER PERIPHERALS, INC. has 101 employees. A table used in the payroll program is referenced by em-

ployee number 1 – 101. Each entry of the table consists of data on the employee as shown in Fig. 13-2. The following program picks up the employee name from the table and moves it into a print buffer.

```
GETNME   EQU    $           EMPLOYEE ID IN A
         LD     HL,TABLE    START OF EMPLOYEE TABLE
         DEC    A           CHANGE ID # TO INDEX 0-100
         LD     B,A         INDEX TO B
         LD     C,0         BC NOW HAS INDEX * 256
         SRL    B
         RR     C           BC NOW HAS INDEX * 128

         ADD    HL,BC       TABLE + (INDEX * 128)
         LD     BC,15       # CHARACTERS IN LAST NAME
         LD     DE,BUFFER   PRINT BUFFER ADDRESS
         LDIR               TRANSFER LAST NAME
```

The ID # (in A) is first converted to an index # of $0\text{-}100_{10}$. The index is then multiplied by 128 as each entry starts at a 128-byte block. When this *displacement* is computed, it is added to the start of the table to find the start of the last name of the employee. An LDIR is then used to move the data into the print buffer.

Suppose that the entries in the above example were not in any order or not indexed by ID number. If the ID # was 1 – 255, then a search through the table could be made for a given ID # key by using the CPI or CPD instruction. The ID # is located in the last byte of each 128-byte entry in the table (see Fig. 13-2). Searching for the ID # from end of the table back would proceed as follows:

```
GETID    LD     HL,TABLE+128*NENT-1   LAST ENTRY+127
         LD     BC,NENT     BYTE COUNT = # OF ENTRIES
         LD     A,KEYID     LOAD KEY ID #
         LD     DE,-127     DECREMENT VALUE
LOOP     CPD                SEARCH FOR ID ONCE
         JR     Z,FOUND     GO IF FOUND
         JP     PO,NFND     GO IF AT END (NOT FND)
         ADD    HL,DE       DECREMENT HL
         JP     LOOP        CONTINUE
FOUND                       HL POINTS TO ID # + 1

NENT     EQU    101         NUMBER OF ENTRIES
```

In the previous program, the HL register was set up to point to the address of the last entry of the table + 127. Since each entry is 128 bytes long, total length of the table is TABLE + (NENT * 128). NENT is the number of entries in the variable length table. The

value TABLE + (NENT * 128) points to the last byte of the table plus one, so one is subtracted from the expression to point to the last ID # in the table. A search is then made, one ID at a time, by a CPD. If a match is not found, 127 is decremented from HL to point to the next ID (the HL register pair has already been decremented once by the CPD). If the ID # is not in the table, BC is decremented down to zero and the NFND routine is entered.

The preceding routine shows a search for a 1-byte key in a table of entries. Searches for n-byte keys may be made using the same methods to sequence through a table, forward or backward, and may use the string comparison techniques described in previous examples.

When it becomes necessary to order tables, the block transfer routines may be used to advantage. Although it is good programming practice not to move large blocks of data from one set of locations in memory to another, at times data must be inserted in tables, deleted from tables, or the key value must be modified and the table reordered. In the following routines, the general parameters are TABLE, the starting address of the table, NENT, the number of entries in the table, and LENT, the length in bytes of each entry. The address of the last word in the table + 1 is given by LASTW. The inputs to the multiply routine are in the A register (multiplicand) and B register (multiplier). The output of the multiplier routine is a 16-bit product in the BC register. The key for the search is always assumed to be in the first byte of the table entry.

The first routine in this set deletes an entry from the table. A search is first made to find the entry by a CPI. When the entry is found, the entries below the delete entry are moved up by an LDIR instruction. Note that this instruction must be an LDIR as an LDDR would overwrite data to be moved as the move was implemented.

```
DELETE  LD      HL,TABLE    START OF TABLE
        LD      BC,NENT     # OF ENTRIES IN TABLE
        LD      A,KEY       SEARCH KEY
        LD      DE,LENT     # OF BYTES PER ENTRY
        DEC     DE          # OF BYTES PER ENTRY - 1
LOOP    CPI                 SEARCH ONE ENTRY
        JP      Z,FOUND     GO IF FOUND
        JP      PO,NFND     NOT FOUND!
        ADD     HL,DE       POINT TO NEXT ENTRY
        JP      LOOP        CONTINUE
FOUND   DEC     HL          POINT TO 1ST BYTE OF DELETE
        PUSH    HL
        LD      HL,LASTW    LAST WORD + 1
        OR      A           CLEAR C
```

```
        POP     BC      # OF BYTES TO MOVE + LENT
        SBC     HL,BC
        OR      A
        SBC     HL,DE   # OF BYTES TO MOVE + 1
        DEC     HL      # OF BYTES TO MOVE
        PUSH    BC
        PUSH    HL
        POP     BC      TRANSFER TO BC
DESTA   POP     DE      DESTINATION ADDRESS
        LD      HL,LENT # OF BYTES PER ENTRY
        ADD     HL,DE   SOURCE ADDRESS
        LDIR            MOVE DATA
DONE    ~
        ~
        ↓
```

In the above routine, a search was first made to find the entry. If found, the remainder of the routine is concerned with setting up the parameters for the move. The HL register points to the destination address + one at this point, so it is decremented and saved in the stack, eventually (at DESTA) to be put into DE. The number of bytes to move is then given by LASTW – HL – LENT, and this value is computed and put into BC. The source address is provided by HL + LENT; this is computed and loaded into HL for the LDIR. The actions of this move are shown in Fig. 13-3.

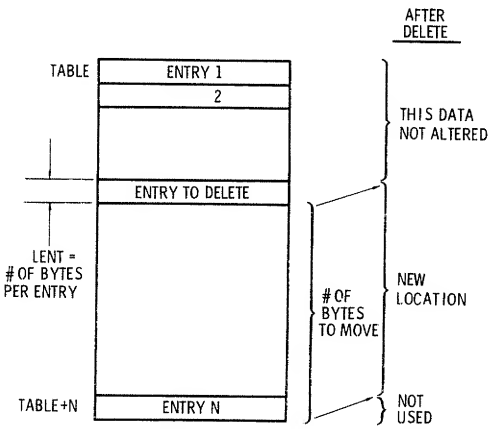


Fig. 13-3. Delete table entry actions.

The next routine inserts an entry into a table. Here, a search is not made on the basis of finding a value equal to a key value, but on finding the two adjacent entries that are less than and greater than the key value (or less than or equal, or greater than or equal). All of the search instructions set the sign flag on each iteration of the

search, so that this comparison may be easily made. When this kind of search is made, a CPI or CPD must, of course, be used as the search is only terminated on equality.

```
INSERT  LD    HL,TABLE   START OF TABLE
        LD    BC,NENT    # OF ENTRIES IN TABLE
        LD    A,KEY      SEARCH KEY
        LD    DE,LENT    # OF BYTES PER ENTRY
        DEC   DE         # OF BYTES PER ENTRY - 1
LOOP    CPI              SEARCH ONE ENTRY
        JP    M,LTHAN    GO IF ENTRY GT KEY
        JP    PO,END     NOT FOUND
        ADD   HL,DE      POINT TO NEXT ENTRY
        JR    LOOP       CONTINUE
LTHAN   DEC   HL         POINT TO GT ENTRY=INSERT POINT
```

A search is first made through the table forwards for the first value that is *greater than the search key.* If no value is found greater than the search key, then the new entry must be *appended to* the table to retain the ascending order of the table. The code for this would be at END and is not shown. If a greater than value is found, then the pointer value in HL minus one defines the source starting address for the move. The destination address is the length of one entry plus the source starting address. The number of bytes to be moved is the total number of table entries minus the current entry multiplied by the number of bytes per entry. The move must be a LDR to prevent overwriting data that has not yet been moved. The actions of the INSERT are shown in Fig. 13-4. Note that this insert is correct even if the new entry will be inserted as the first entry of the table.

The third routine of this set modifies the key value of a table entry. As the new value disturbs the order of the table, the table must be reordered. One way to implement a modify of this kind would be to

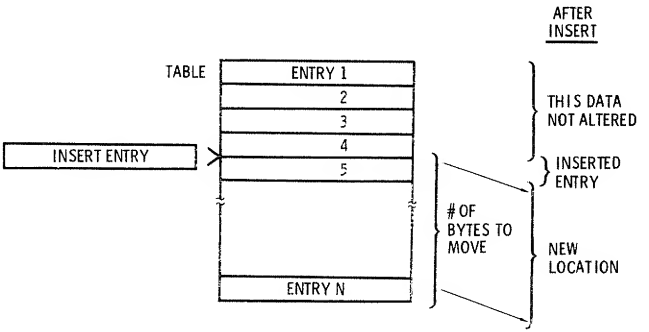


Fig. 13-4. Insert table entry actions.

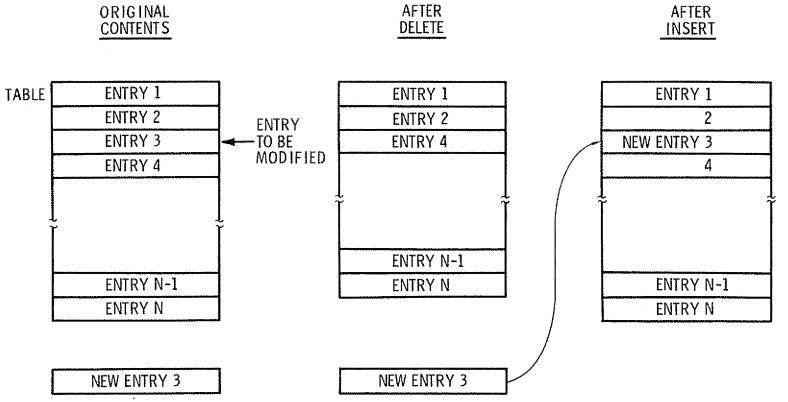


Fig. 13-5. Modify table entry actions.

delete the entry using the delete routine, and then to insert the modified entry using the insert routine. The actions of the MODIFY are shown in Fig. 13-5.

The search instructions lend themselves to *sequential* searches through tables of data where each data entry is accessed in sequence while moving through the table forward or backwards. Various other search algorithms are possible for ordered data entries. A binary search accesses table entries by comparing the sense of the (key: table) entry comparison. In a table ordered with entries in ascending order, the next entry accessed will be the middle entry of the remaining entries *before* the current entry, if the current entry is greater than the key value; or the middle entry of the remaining entries *after* the current entry, if the current entry is less than the key value. The algorithm converges on the sought entry in $N = (\log_2 NENT) + 1$, where NENT is the number of entries in the table, as shown in Fig. 13-6. A table of 1000 entries can be searched for a

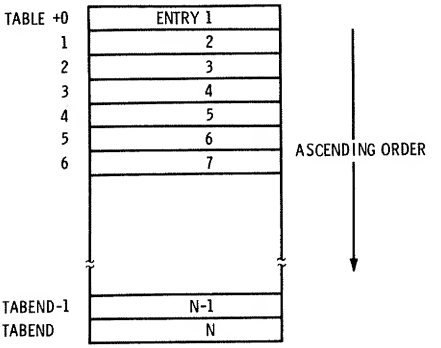


Fig. 13-6. Table for binary search example.

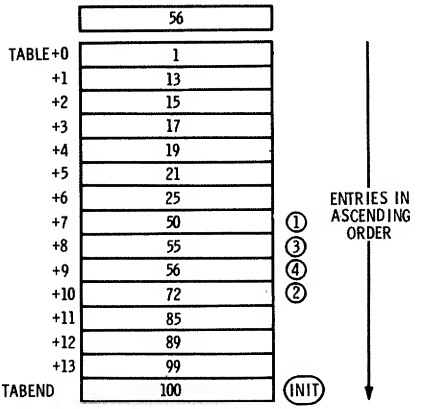
given entry in 11 iterations or less [$N = (\log_2 1000) + 1 = 9.XX + 1 = 11$], whereas a sequential search will take an average of 500 accesses.

The following routine performs a binary search for an 8-bit value for a table of 256 entries, or less, that starts at TABLE and ends at TABEND. Each entry has one byte, the search key itself, as shown in Fig. 13-6.

```
BINSRC  LD    B,0              SET LOW INDEX = 0
        LD    C,TABEND-TABLE   SET HIGH INDEX = LAST
        LD    D,0              INITIALIZE DE
        LD    E,C              INITIALIZE LAST INDEX
        LD    IX,KEY           POINT TO KEY
        JP    JUMP1            FIRST TEST = LAST ENTRY
LOOP    LD    A,C
        SUB   A,B              HIGH-LOW
        SRL   A                (HIGH-LOW)/2
        LD    E,A              MIDPOINT DISPLACEMENT
JUMP1   LD    HL,TABLE         TABLE START
        ADD   HL,DE            START + (HIGH-LOW/2)
        LD    A,L
        ADD   A,B              LOW + (HIGH-LOW/2)
        LD    L,A
        JP    NC,JUMP2         GO IF NO CARRY TO MSB
        INC   H                BUMP MSB
JUMP2   LD    A,(HL)           GET ENTRY
        CP    (IX)             COMPARE WITH KEY
        JP    Z,FOUND          GO IF MATCH
        JP    M,JUMP3          GO IF ENTRY LT KEY
        LD    B,E              CURRENT TO LOW
        JP    LOOP             CONTINUE
JUMP3   LD    C,E              CURRENT TO HIGH
        JP    LOOP             CONTINUE
```

The routine, first of all, compares the last entry of the table with the search key (to avoid *truncation* errors in computing the next index). If the key is not found, the iterative portion of the routine is entered. For each iteration, new low and high limits are established based on the results of the last comparison. The current index in E is put into either C (high) or B (low) as the new limit. The midpoint displacement of the area to be searched is then computed (HIGH − LOW)/2, and added to the address of low to find the next location for the search. The value in the next location is then compared with the search key and if unequal, the next binary search area and location are computed. In the above loop, there is no check on terminating conditions for the search. If the key exists, as we have assumed, it will eventually be found and the instruction at FOUND will be executed. If the key does not exist, no termination of the loop

will occur. To prevent an endless loop, a simple comparison should be made of the (i+1) index value [(HIGH − LOW)/2] to the ith index value. When the two values are equal, the binary search has ended without a match. The binary search actions of this routine for a 15-entry table are shown in Fig. 13-7.



ITERATION	LOW INDEX	HIGH INDEX	ENTRY ACCESSED	SENSE OF MATCH
INITIALIZATION	TABLE+0	TABLE+14	TABLE+14	>
1	+0	+14	+7	<
2	+7	+14	+10	>
3	+7	+10	+8	<
4	+8	+10	+9	=

Fig. 13-7. Binary search example.

LIST OPERATIONS

When sorted data must be altered frequently and entries deleted, inserted, or modified, the overhead of altering data tables becomes significantly large. A list is a data structure that reduces the overhead by eliminating movements of large blocks of data when items are changed within the list. A single-ended list consists of entries that are noncontiguous in memory. Each entry consists of the data associated with that list entry and a pointer (address) to the next data items in the list. As the data items are linked by the address pointers, the entries in the list may be in any order in memory. The head of the list is referenced by a variable in memory. The last item in the list often has a −1 or other invalid address to signify that it is the last item. Fig. 13-8 shows a typical single-ended linked list of nine data items with four bytes per entry. The last two bytes are the pointer to the next item in the list. The head of the list is referenced by HEADLS.

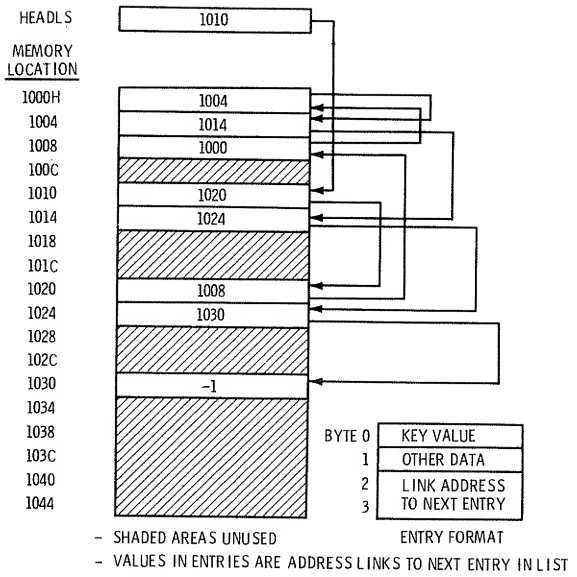


Fig. 13-8. Typical single-ended linked list.

When data is to be deleted from the list, the previous pointer is simply changed to the link address of the deleted item. When data is to be inserted the pointer of the data item before the insertion point is changed to the address of the new item. The address of the new data item is loaded with the link address of the data item before the insertion point. These actions are illustrated in Fig. 13-9.

The following Z-80 code shows a search of a single-ended linked list for a given search key value. Each data item of the list consists of an 8-bit data value and a 2-byte link address.

```
INSERT  LD    HL,HEADLS   HEAD OF LIST
        LD    A,KEY       SEARCH KEY
        LD    BC,1        FOR END OF LIST COMPARISON
                            OR A RESET CARRY
LOOP    ADC   HL,BC       NEXT ADDRESS + 1
        JP    Z,END       END OF LIST, NOT FOUND
        DEC   HL          NEXT ADDRESS
        LD    D,(HL)      GET NEXT ENTRY
        CP    D           COMPARE NEXT TO S KEY
        JP    Z,FOUND     GO IF FOUND
        INC   HL          POINT TO ADDRESS
        PUSH  HL          TRANSFER HL TO IX
        POP   IX
        LD    H,(IX+1)    GET MSB OF ADDRESS
        LD    L,(IX)      GET LSB OF ADDRESS
        JP    LOOP        CONTINUE
```

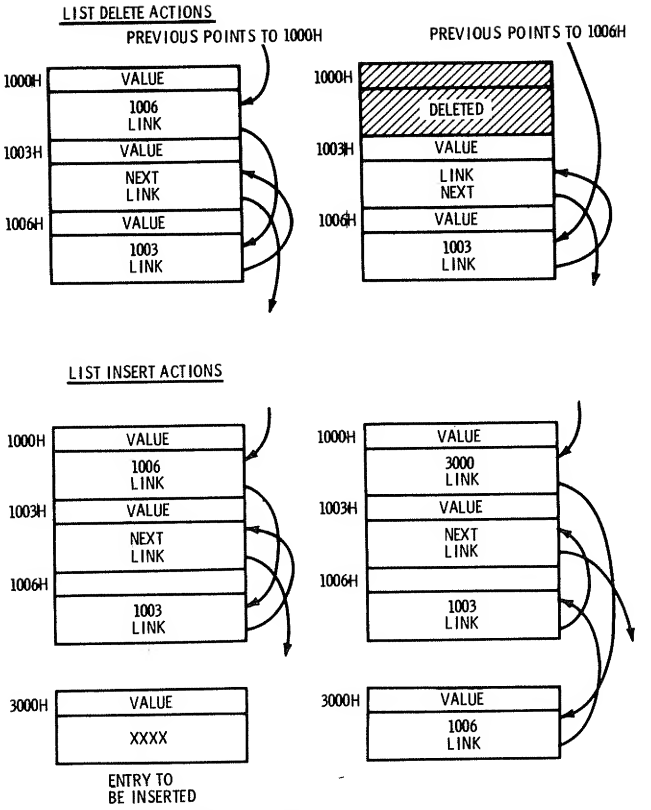


Fig. 13-9. List delete and insert actions.

The address of the first entry of the list is loaded from HEADLS. A check is made to see if the address of the next data item is a −1 which would signify the end of the list, or an empty list (HEADLS = −1). The check is made, incidentally, by adding one to the address in HL by addition of BC to HL. The ADC HL,SS and SBC HL,SS are unique in that they set the Z flag if the result is zero; none of the other 16-bit arithmetic instructions affect the Z flag. If the address is a valid address, the data value from the next list entry is retrieved and a match is tried. If no match is found, the link address of the data item is loaded into HL in preparation for the next comparison.

List operations for inserts and deletes are similar to the sequential searches discussed under table operations. More overhead is involved in finding the insertion or deletion point for the list, but once the proper list entry is found, only a slight additional amount of code is involved to change pointers and the data movement over-

head is avoided. The following code illustrates insertion of a new data item in a list after the data item pointed to by IX.

```
INSERT  LD    B,(IX+2)          GET LINK ADDRESS
        LD    C,(IX+1)
        LD    A,(NEWLKA)        NEW LS ADDRESS BYTE
        LD    (IX+1),A          REPLACE ITEM (M - 1) ADDRESS
        LD    A,(NEWLKA+1)
        LD    (IX+2),A
        LD    IY,NEWLKA         NEW LINK ADDRESS
        LD    (IY+2),B          STORE NEXT LINK
        LD    (IY+1),C            ADDRESS
```

At entry, IX points to the list entry immediately preceding the insertion point. The link address of this list entry is picked up in BC. This link address will point to the list entry which will be immediately after the inserted entry. The address of the new data item is in the 2-byte location NEWLKA, and NEWLKA+1. This address is stored in the link address bytes of the previous list entry. Finally, the link address of the list entry which will precede the inserted entry is stored in bytes one and two of the insert entry.

The preceding paragraph briefly discusses list operations using the Z-80 instruction set. More sophisticated list operations such as double-ended lists are commonly used but the techniques involved are somewhat similar to the lists described.

CHAPTER 14

Subroutine Operation—Jump, Call, and Return Groups

The instructions in these groups allow the user to jump conditionally, or unconditionally, to locations within the relative addressing range or to jump directly with extended addressing to any memory location. If the jump is to a memory location and a return is not to be made back to the instruction following the jump, then the jump is a JP or JR. If the jump saves the return address in the stack, then the jump is a CALL. CALLs are used to transfer control to subroutines, which are simply segments of code designated as subroutines that occur in one place in memory. These segments can be used by many different parts of the program, avoiding duplication of the code at every point where the functions of the subroutine are to be performed. Each subroutine is entered by a CALL, which saves the return address of the instruction following the CALL in the stack; each subroutine is terminated by an RET, or return. Returns may be conditional, or unconditional. If the return is conditional, it is based on the condition-code settings just as a conditional jump is. The RST or restart instruction is a special CALL that can be used for page zero subroutine call use or for interrupt responses.

JUMP INSTRUCTIONS

The unconditional jump instruction, JP, has been used in many examples in previous chapters and should be no problem to the

reader. When a choice must be made between a JP and an unconditional relative jump JR, the relative jump should be chosen if the jump address is within the relative addressing range of the instruction. The JR uses two bytes, as opposed to the three bytes of the JP. (The JP *is* faster, however, 2.5 microseconds compared to 3.0 microseconds for the JR.) The relative addressing range of the JR is up to 126 locations back from the JR or up to 129 locations forward from the JR. The assembler used for the Z-80 will give a diagnostic message on the assembly listing, if this range is exceeded, and a JP can then be substituted for the JR. The JR will be possible in most of the jump cases.

The conditional extended-addressing jump JP and relative-addressing jump JR enable a jump to a location based on the state of the zero flag, carry flag, parity flag, or sign flag as shown in Table 14-1.

Various assemblers will use different mnemonics for the condition. NZ, Z, NC, C, P, and M are self-explanatory, but the PO and PE mnemonics may be replaced with more descriptive mnemonics such as V or NV for overflow and no overflow; and P and NP for parity or no parity.

Table 14-1. Conditional Jumps

Flag	Extended Form	Relative Form
NZ nonzero	JP NZ,LOCN	JR Z,LOCN
Z zero	JP Z,LOCN	JR NZ,LOCN
NC no carry	JP NC,LOCN	JR C,LOCN
C carry	JP C,LOCN	JR NC,LOCN
PO parity odd	JP PO,LOCN	none
PE parity even	JP PE,LOCN	none
P sign positive	JP P,LOCN	none
M sign negative	JP M,LOCN	none

Three jump instructions JP (HL), JP (IX), and JP (IY) effectively cause an unconditional jump by loading the program counter with the contents of the HL, IX, or IY registers. When a multiple path decision must be made and a jump to one of several points is effected, these instructions may be used to advantage. As an example, suppose that in a large program we have a "mode" word that indicates what mode the system is in currently. This could be used to indicate the pass number of a three-pass assembler or any combination of conditions the programmer desires. For our purposes, let us assume the mode word MODE represents states of the system as shown in Fig. 14-1. Note that some combinations of states are not possible, but that the general range for the four bits of the mode

"MODE" WORD STATE ENCODING

FOR MICROPROCESSOR CONTROLLED AUTOMAT:

OTHER BITS				MODE				
X	X	X	X	0	0	0	0	NO SANDWICH
X	X	X	X	0	0	0	1	PEANUT BUTTER SANDWICH
X	X	X	X	0	0	1	0	JELLY SANDWICH
X	X	X	X	0	0	1	1	PEANUT BUTTER & JELLY SANDWICH
X	X	X	X	0	1	0	0	NOT POSSIBLE
X	X	X	X	0	1	0	1	PEANUT BUTTER, BANANA SANDWICH
X	X	X	X	0	1	1	0	JELLY AND BANANA SANDWICH
X	X	X	X	0	1	1	1	PEANUT BUTTER, JELLY, BANANA SANDWICH
X	X	X	X	1	0	0	0	NOT POSSIBLE
X	X	X	X	1	0	0	1	PEANUT BUTTER, HOLD THE BREAD
X	X	X	X	1	0	1	1	PEANUT BUTTER AND JELLY, HOLD THE BREAD
7	6	5	4	3	2	1	0	

X = DON'T CARE

Fig. 14-1. Mode word example.

word is from 0 through 11. The following code takes the mode word and "jumps out" based on a jump, or branch table, to the proper processing routine for the current system mode.

```
TESTMD   LD    BC,JUMPTB    JUMP TABLE ADDRESS
         LD    A,MODE       GET MODE WORD
         AND   A,FH         STRIP OFF MODE BITS
         LD    L,A
         LD    H,0          MODE TO HL
         ADD   HL,HL        MODE * 2
         ADD   HL,BC        JUMPTB + (MODE * 2)
         JP    (HL)         JUMP OUT
JUMPTB   JR    MODE0
         JR    MODE1
         JR    MODE2
         JR    MODE3

         JR    ERROR        NOT POSSIBLE
         JR    MODE5
         JR    MODE6
         JR    MODE7
         JR    ERROR        NOT POSSIBLE
         JR    MODE9
         JR    MODE10
         JR    MODE11
```

The mode field is first multiplied by two and then added to the address of the jump table. Each entry in the jump table is two bytes long, so the multiplication by two indexes into the table properly, causing a jump by the JP (HL) to the jump in the table correspond-

ing to the mode. The alternative to the jump table would be coded such as the following:

```
LD    A,MODE     GET MODE WORD
AND   A,FH       STRIP OFF MODE BITS
JR    Z,MODE0    GO IF MODE 0
CP    A,1
JR    Z,MODE1    GO IF MODE 1
CP    A,2
```

The jump table approach is somewhat cleaner to implement and for a large number of combinations is more efficient in memory storage and execution-time requirements.

The most interesting instruction in the jump group is the DJNZ instruction, a 2-byte relative-addressing instruction. The DJNZ is a decrement and jump if nonzero-type instruction. The contents of the B register is decremented by one. If the result is nonzero, a jump is made to the address specified. If the result of the decrement is zero, the next instruction in sequence is executed. This instruction replaces the code:

```
DEC   B          DECREMENT COUNT IN B
JR    NZ,LOOP    GO IF NOT AT END
```

In any case, where an iterative routine is implemented with a loop count of 256 or less, the DJNZ may be used to advantage. Using one of the previous examples as an illustration, here is a typical use of DJNZ.

```
PARITY  XOR   A          CLEAR PARITY AND C
        LD    B,8        INITIALIZE COUNT
        LD    HL,MEMOP   MEMORY OPERAND ADDRESS
LOOP    RLC   (HL)       SHIFT OUT BIT TO CY
        JR    NC,JUMP1   GO IF NOT A ONE BIT
        XOR   1          FLIP PARITY INDICATOR
JUMP1   DJNZ  LOOP       GO IF NOT 8 BITS
DONE                     A REGISTER NOW 0 IF EVEN
                           # OF 1 BITS, 1 IF NOT
```

The DJNZ executes in 3.25 microseconds for the jump and 2.0 microseconds if the result is 0. The equivalent DEC and JR execute in 4.0 and 2.75 microseconds for the jump and no jump, respectively, and use one more byte in memory.

SUBROUTINE USE

A subroutine is CALLed by a CALL instruction and terminated by a RETurn instruction. A subroutine almost always executes a

return instruction to properly pop the return address from the stack. If this is not done, subsequent pops will pull the wrong data from the stack. The only other alternatives to executing a return is to reinitialize the stack by an LD SP instruction for some catastrophic system malfunction, or to increment the stack pointer by two to "reset" the SP to the point at which the return would have left it. The latter procedure is poor programming practice although it can easily be done by:

```
ADJUST    INC  SP      DO NOT RETURN
          INC  SP        TO CALLING PROGRAM
```

We have seen subroutine use in previous examples. The simplest case is a subroutine that performs one function only. No parameters are passed to the subroutine and the subroutine executes its predefined function and returns to the calling program. An example of this is the code below which shifts the DE register pair one bit position to the right in a logical shift.

```
SHRL      SRL  D     SHIFT HIGH ORDER
          RR   E     SHIFT LOW ORDER
          RET        RETURN TO CALLING PROGRAM
```

In many cases, however, subroutines will perform a function that requires a parameter or number of parameters defining the function. A simple example of this would be the more functional subroutine that follows which shifts the DE register logically right a specified number of times. The *argument* N is *passed* in the B register.

```
MSHRL     CALL  SHRL    SHIFT DE ONCE
          DJNZ  MSHRL   CONTINUE N TIMES
          RET           RETURN TO CALLING PROGRAM
```

When there are many arguments to be passed to the subroutine, there are a number of solutions. The contents of the CPU registers

```
          PUSH  AF          SAVE A PARTIAL RESULT
          PUSH  BC          SAVE BC
          PUSH  DE          SAVE DE
          PUSH  IX          SAVE BUFFER ADDRESS
          LD    A,MODE      SYSTEM MODE
          LD    BC,SOURCE   I/O BUFFER ADDRESS
          LD    DE,COMLST   I/O COMMAND LIST
          LD    IX,ERADR    ERROR MSG ADDRESS
          CALL  WRTAPE      WRITE OUT TAPE RECORD
          POP   IX          RESTORE
          POP   DE            ALL
          POP   BC              REGISTERS
          POP   AF
```

may be temporarily saved in the stack, and then used as parameter storage before the subroutine is called. Once the subroutine action has been completed, the original register contents can be retrieved from the stack.

Another way of passing a number of arguments is to put the arguments (or parameters) into a parameter list in memory. The address of the list can then be loaded into one of the register pairs, or index registers, and passed to the subroutine, which can then easily pick up the arguments.

```
LD      IY,IOBLK      LOAD PARAMETER LIST
CALL    WRTAPE        WRITE OUT TAPE RECORD
```

The following subroutine, BXOAS, converts a 16-bit binary value in the DE register to a string in octal digits and stores those digits in a specified buffer area. The parameters to be passed are defined in a calling sequence that is nothing more than a description of the parameters used in the subroutine, how they are passed, and how the subroutine is used. Many times it is convenient to define the calling sequence in the assembly code itself as follows:

```
;*************************************************
;* SUBROUTINE BXOAS                              *
;*                                               *
;* FUNCTION: THIS SR CONVERTS A 16-BIT           *
;*      BINARY NUMBER IN D,E TO A SIX-DIGIT      *
;*      ASCII CHARACTER STRING.                  *
;* CALLING SEQUENCE:                             *
;*          (H,L) = POINTER TO CHARACTER BUFFER + 5 *
;*          (D,E) = BINARY #                     *
;*          CALL BXOAS                           *
;*          (RETURN WITH CHARACTERS              *
;*            CONVERTED, (HL) = FIRST CHARACTER  *
;*            POSITION IN BUFFER - 1, ALL OTHER  *
;*            REGISTERS SAVED                    *
;* ERROR CONDITIONS: NONE                        *
;*************************************************
BXOAS   PUSH  DE        SAVE NUMBER
        PUSH  AF        SAVE A, FLAGS
        PUSH  BC        SAVE BC
        LD    C, 6      LOAD ITERATION COUNT
LOOP    LD    A, 7      MASK
        AND   A, E      GET CURRENT OCTAL #
        ADD   A, 30H    CONVERT TO OCTAL ASCII
        LD    (HL), A   STORE
        DEC   HL        BUMP CHARACTER PNTR
        DEC   C         DECREMENT ITERATION CNT
        JR    Z, DONE   GO IF DONE
```

```
        LD      B, 3        FOR SHIFT SUBROUTINE
        CALL    MSHRL       SHIFT DE 3 PLACES RIGHT
        JP      LOOP        CONTINUE
DONE    POP     BC          RESTORE ALL
        POP     AF            REGISTERS AND
        POP     DE              RETURN
        RET
```

The preceding subroutine illustrates another important aspect of subroutine use. It is important to know which registers the subroutine uses and which registers are saved. BXOAS saves all registers used, such as DE (shifted and masked), A (used for conversion), and C (used for iteration counting). In addition, BXOAS calls another subroutine, MSHRL, which uses the B register, so that the B register must also be saved. The preceding sequence, by the way, uses three levels of subroutines. BXOAS calls MSHRL, which calls SHRL. This *nesting* of subroutines can be extended to as many levels as convenient. Stack operations are automatic and create no problems as long as there is a return for every call and a pop for every push.

All of the above examples have dealt with unconditional CALLs and RETurns, but conditional calls and returns may also be made. The conditions and mnemonics for these are the same as for the conditional extended jumps, as shown in Table 14-2.

The uses of the conditional calls and returns are identical to the applications of their unconditional counterparts. In the following example, an argument is loaded into A and subroutine ABSVAL is called if the number is negative. A bit test instruction tests the sign bit to reset the Z flag if the number is negative.

```
LD      A, (IX)         LOAD FIRST ARGUMENT
BIT     7, A            TEST SIGN
CALL    NZ, ABSVAL      TAKE ABS VAL IF NECESSARY
```

An example of a conditional return is in the following routine which finds the integer portion of the square root of an 8-bit number.

Table 14-2. Conditional Calls and Returns

Flag	Call	Return
NZ	CALL NZ,SRTN	RET NZ
Z	CALL Z,SRTN	RET Z
NC	CALL NC,SRTN	RET NC
C	CALL C,SRTN	RET C
PO	CALL PO,SRTN	RET PO
PE	CALL PE,SRTN	RET PE
P	CALL P,SRTN	RET P
M	CALL M,SRTN	RET M

The number is in the A register on entry and the integer portion of the square root is in B on exit.

```
SQRT    LD    B, 0      INITIALIZE SQ ROOT
        LD    C, 1      INITIALIZE FIRST ODD #
LOOP    SUB   A, C      SUBTRACT NEXT ODD INTEGER
        RET   M         RETURN IF DONE
        INC   C
        INC   C         BUMP TO NEXT ODD INTEGER
        INC   B         BUMP PARTIAL SQ ROOT
        JP    LOOP      CONTINUE
```

None of the instructions in the jump, call, or return groups alter the condition codes when a jump is made to another memory location, or subroutine. This means that a subroutine may *pass back* parameters in the condition-code flags themselves as in the following example where the carry from a 4-byte add is passed back on exit from the subroutine, in addition to the other flags.

```
ADD4    LD    BC, (ARG1 + 2)    LS 2 BYTES OP 1
        LD    HL, (ARG2 + 2)    LS 2 BYTES OP 2
        ADD   HL, BC            OP 1 + OP 2 LS
        LD    (ARG1 + 2), HL    STORE RESULT LS
        LD    BC, ARG1          MS 2 BYTES OP 1
        LD    HL, ARG2          MS 2 BYTES OP 2
        ADC   HL, BC            OP 1 + OP 2 MS
        LD    (ARG1), HL        STORE RESULT MS
        RET                     RETURN WITH C, Z, P,
                                  SET
```

The RST, or restart instruction, is rather a "leftover" from the 8080 implemented for compatibility. The NMI and mode 1 interrupt capability serves small microcomputer configurations well, and the mode 2 interrupt capability is excellent for larger configurations with many interrupts of different levels.

If the RST is not to be used in external interrupt circuitry (mode 0), then it may be used as a special 1-byte call. The RST performs the same actions as a call, but allows a jump to only one of eight page zero locations, 00H, 08H, 10H, 18H, 20H, 28H, 30H, or 38H. The advantage of using the RST is that if frequently called subroutines are vectored from page 0 some memory will be saved as the 1-byte RSTs replace 3-byte calls. The stress here is on "frequently called." If 100 parameter calls are made to a subroutine in the course of a program, then 200 locations in memory will be saved, a significant savings for minimum configuration systems. Although the eight locations available for each CALL limits the subroutine action quite severely, some subroutines may be implemented in eight bytes and

in others the page 0 location may simply hold an extended addressing JP to another memory area. If all of the above is worth the trouble, typical RSTs would appear as follows.

```
                ORG     0
                LD      (HL), A         STORE A
                INC     HL              INCREMENT
                RET
                ORG     08H
                LD      A, (HL)         LOAD A
                INC     HL              INCREMENT
                RET
                ORG     10H
                RLCA                    ROTATE A 4 LEFT
                RLCA                        ~
                RLCA                        ~
                RLCA                        ~
                RET                         v
                ~
STRING          EQU     0
LDINC           EQU     08H
R04LF           EQU     10H
                ~
                v
                RST     LDINC           LOAD BYTE 1
                RST     STRING          STORE
                RST     LDINC           LOAD BYTE 2
                RST     STRING          STORE
                RST     LDINC           LOAD BYTE 3
                RST     R04LF           ROTATE 4
                AND     A, FH           STRIP BCD DIGIT
```

The three routines at locations 0, 08H, and 10H are typical of the commonly used routines that could be placed in page 0. The call is made to the proper page 0 location by the RST with an argument previously equated to the page 0 location. RST LDINC would be identical to RST 8, for example.

REENTRANCY

The subroutine calls, returns, and stack instructions facilitate the writing of *reentrant* code. Reentrancy in a portion of code means that the code may be reentered due to interrupt processing. Reentrancy is no problem if the environment is saved when the interrupt is received and if in the routine that is reentered, no common memory locations are altered. If common memory is altered, then reentrancy may destroy the results or partial results of the previous users of the reentered routine.

Consider the following example. The routine below stores the contents of the A register in temporary storage location in memory called TEMP1. No interrupts are active. Several instructions after the store, an interrupt occurs. As the interrupts are enabled, the interrupt is acted on and the interrupt processing routine is entered. During the course of the interrupt processing, the interrupt processing routine calls subroutine FINDIT. As the subroutine was being executed when the interrupt occurred, it is reentered. During the course of the routine, a new value is stored in TEMP1. Later, the subroutine is executed and a return made back to the interrupt processing routine. Eventually, the interrupt processing routine finishes, restores the environment, and executes a RETI or RETN to return back to the interrupted point, in this case location BACKHR. At the next instruction, the contents of the A register is reloaded, but the value reloaded is the value stored for the second entry of FINDIT, not the first! Reentrancy has destroyed the previous contents of TEMP1.

```
REENTRY ——→ FINDIT
                        LD      (TEMP1), A      SAVE A TEMP
                        LD      A, (HL)         GT NXT PARAM
                        LD      B, (IX + 30H)   GT 2ND VALU
                        ADD     A, B
INTERRUPT →             LD      (HL), A         STORE RESULT
ENTER HERE → BACKHR     LD      A, (TEMP1)      RESTORE
  AFTER
INTERRUPT
PROCESSING
                        RET                     RETURN
            TEMP1       DEFS    1               TMPRY STRG
            TEMP2       DEFS    1
```

There are many ways around reentrancy. The easiest is to never alter common memory locations within a subroutine. The stack actions for storage will automatically save parameters and provide almost unlimited temporary storage of variables. With stack storage, a subroutine can be reentered as many times as practical while still preserving temporary storage for each entry level. If it becomes necessary to use areas of memory for storage, then reentrancy is still possible if there is a separate user area for each level of reentrancy. In a simple example of this, let us say there are five users that cause five separate interrupts. The interrupt processing for each of the interrupts calls subroutine GETCH that reads the next available character from a keyboard input and stores it in the next buffer loca-

tion for the user. The interrupt processing routine would call the subroutine with a user number.

```
LD      A,3         THIS IS USER # 3
CALL    GETCH       LOOK FOR NEXT CHARACTER
OR      A           SET FLAGS
JP      NZ,AVAIL    GO IF CHARACTER AVAILABLE
```

The subroutine at GETCH would utilize the user number to find the predefined user area (or task block) for storage of data and variables if required.

```
GETCH   PUSH    AF          SAVE USER #
        SLA     A           (USER #) * 2
        LD      C,A         (USER #) * 2 IN BC
        LD      B,0
        LD      HL,BUFTB    TABLE OF BUFFER ADDRESSES
        ADD     HL,BC
        LD      E,(HL)
        INC     HL
        LD      D,(HL)      GET BUFFER ADDRESS
        CALL    READK       READ KEYBOARD
        JP      Z,OUT       GO IF NONE AVAILABLE
        LD      (HL,)A      STORE CHARACTER

BUFTB   DEFW    BUFF0       TABLE OF BUFFER
        DEFW    BUFF1         ADDRESSES USERS
        DEFW    BUFF2           0 THRU 4
        DEFW    BUFF3
        DEFW    BUFF4
```

Other ways to avoid the reentrancy problem are "lock-outs" of subroutines in use, disabling of interrupts during critical sequences, and replication of code for each level of user, but the above techniques are probably the most common and the use of stack for temporary storage is the cleanest implementation.

CHAPTER 15

I/O and Interrupt Operations— I/O and CPU Control Groups

The I/O instructions allow the Z-80 system user to input or output data a byte at a time under programmed I/O. Compatibility with the 8080 is provided in the Z-80 IN A,(N) and OUT (N),A instructions which transfer data by means of the A register only. Data may be transferred between *any* general-purpose CPU register and the I/O device controller with the IN R,(C) and OUT (C),R instructions, however. I/O block transfer instructions allow semi-automatic or automatic transfer of up to 256 bytes of data with an operation similar to the other block-oriented instructions.

The interrupt actions in the Z-80 are controlled by the interrupt enable, disable instructions and by the interrupt mode instructions. Several interrupt modes are possible, depending on system configuration. The maximum interrupt capability of a Z-80 system will handle many levels of interrupts with priority encoding and automatic vectoring.

A REGISTER I/O INSTRUCTIONS

The IN A,(N) and OUT (N),A I/O instructions are downwards compatible with the equivalent IN and OUT 8080 instructions. Both instructions are a 2-byte immediate instruction with the first byte specifying the operation code and the second byte specifying an 8-bit I/O address N from 0 through 255. As described in Chapter 8, when an IN A,(N) instruction is executed, the I/O port address N is placed on address lines A7 through A0. The addressed device con-

troller outputs the data byte to be transferred to the data bus after decoding the address and $\overline{\text{IORQ}}$ (and $\overline{\text{RD}}$). The CPU then strobes the data into the A register. When an OUT (N),A instruction is executed, the address N is output to address lines A7 through A0, as in the case of the IN instruction. Some time later within the OUT instruction cycle, the CPU outputs the contents of the A register onto the data bus and also provides the $\overline{\text{IORQ}}$ and $\overline{\text{WR}}$ signals. The I/O device controller then strobes the 8-bit data into its internal register. Both the IN A,(N) and OUT (N),A instructions output the device address on A7 and outputs the contents of the A register on A15 through A8.

The simplest form of an IN instruction loop goes through the following sequence:

1. Is the next data byte ready from device controller N?
2. If not, go to 1.
3. Input the data byte.
4. Exit the subroutine.

The preceding sequence requires that the device controller know when the next data byte is ready and has some way of communicating the ready status to the CPU. The device controller must also be able to decode the query for ready status and the request to transmit the data byte. A simple way to implement the preceding instruction is to assign one I/O address to the status query port, and one to the I/O data port. Some format for the status must also be established so that the CPU can decode the status in software. Assuming that a Teletype keyboard is being read with a status port address of 1 and a data port address of 0, the routine to read in one byte of data would appear as follows.

```
READC   EQU   $        READ CHARACTER ROUTINE
LOOP    IN    A,(1)    GET STATUS
        BIT   0,A      TEST READY STATUS
        JP    Z,LOOP   GO IF NOT READY
        IN    A,0      GET BYTE
        RET            EXIT
```

The subroutine first reads the status from the Teletype controller. The controller responds by outputting the status byte which in this case is all zeros in D7 through D1 and a 1 in D0 if the next byte is ready. The routine keeps on reading status until the status indicates data ready. After this condition is detected, the CPU reads in the data byte by addressing the data port of the teletypewriter controller (device address 0). The action of reading in the data byte resets the ready status in the teletypewriter device controller. The status remains reset until the next character is typed by the operator and

stored in the teletypewriter controller buffer (no earlier than 100 ms for a typical teletypewriter).

As described in previous chapters, this routine is very much *I/O bound.* Most of the execution time of the routine is spent in the three-instruction loop waiting for the next data byte to appear and set the ready status. The alternative method of interrupt transfers is described later in this chapter. It is one way to overlap processing and I/O. The maximum data transfer rate of the above loop can be easily calculated. The four instructions require a total execution time of 10.0 μs. If a high-speed I/O device was able to transfer one byte every 10 μs, this timing loop would just be able to keep up with it with perfectly synchronized timing. The absolute maximum data transfer rate would thus be 100,000 bytes per second. High-speed I/O devices rarely use a timing loop such as this; most devices that are high-speed, relatively synchronous devices will use DMA, or direct-memory-access. DMA allows the device controller to access external memory independently of CPU processing, and enables overlap of I/O transfers and CPU processing that is transparent to the program being executed. DMA operations are described later in this chapter.

The OUT (N),A instruction functions quite similarly to the IN A,(N) instruction. The output process proceeds as follows: The ready status of the device controller is first tested by reading in the status. If the device controller is ready (done processing the previous character), the program performs an OUT (N),A, outputting the previously loaded A register to the data bus. The output sets the "busy" status in the device controller. When the character has been transmitted from the device controller buffer to the I/O device, the busy status is reset. A typical output subroutine for one character output would appear as shown next.

```
WRITEC  EQU  $          WRITE CHARACTER ROUTINE
LOOP    IN   A,(1)      GET STATUS
        BIT  1,A        TEST READY STATUS
        JP   NZ,LOOP    GO IF NOT READY
        OUT  (0),A      OUTPUT BYTE
        RET             RETURN
```

Here, the status is assumed to be compatible with the input routine, bit 1 of the status indicating that the output section of the device controller has finished writing the last byte. When the device controller indicates "not busy" (bit 1 = 0), the contents of A are output to the data channel of the device.

The above routines are designed to output, or input, one character at a time in simplistic fashion, yet many devices, such as teletypewriters, paper-tape readers and punches, line printers, certain crt

displays, and others require no more sophisticated communication than the above. There may be embellishments with more particulars about device states (parity, device ready, etc.), but the above routines, or ones like them, are quite usable for simple I/O devices.

Prior to the input and output operations above, a *clear* command may have been issued to the I/O device. This is generally done before each block of I/O transfers when the device should not be "hung" in an erroneous busy state to clear the status and initialize data transfers. Depending on the I/O device controller involved, this may range from a simple write of a clear command to the status port, or something more elaborate.

I/O INSTRUCTIONS USING C REGISTER

The remaining nonblock transfer I/O instructions utilize the C register in holding the I/O address, 0 through 255. Data may be transferred to one of the general-purpose CPU registers A, B, C, D, E, H, or L by the IN R,(C) instruction. Data may be transferred from one of the registers to an I/O device by the OUT (C),R instruction. In both cases, the C register must be loaded with the I/O device address. The contents of the C register is output to address lines A7 through A0 *while the contents of the B register* are output to address lines A15 through A8. The B register contents may be used for communication of status or for outputting the current byte count, as in the I/O block transfer instructions to be discussed. They may, of course, also be ignored, as the C register holds the actual device address and the data bus is used to transfer data to and from the I/O device controller. Setup of the registers for these I/O instructions are similar to the previous I/O instructions using the A register.

```
WRITEC  LD    D,(HL)   GET NEXT CHARACTER
        LD    C,30H    DEVICE ADDRESS
LOOP    IN    E,(C)    GET STATUS
        JP    Z,LOOP   GO IF NOT READY
        OUT   D,(C)    OUTPUT DATA BYTE
        RET            RETURN
```

In the preceding routine, the D register was loaded with the data to be output and the C register was loaded with the device address of 3OH. Here the status port address is the same as the data port address, assuming a "write only" device that supplies status for a read. The IN E,(C) instruction inputs status into the E register. One bit of status, ready (1) or not ready (0) was assumed. If the device was ready, the data in the D register was output by the OUT D,(C) instruction.

The IN instruction using the C register is different from the former A register IN instruction in that input data *does* set the condition codes in the C register I/O instruction case. The sign flag, zero flag, and parity flags are all affected by the input data and can be tested after execution of an IN R,(C) instruction, as was done in the previous example.

I/O BLOCK TRANSFER INSTRUCTIONS

The I/O block transfer instructions enable data to be input or output in blocks of 1 to 256 bytes. The block access is either forward through a block, or backwards through a block, as is the case in the other Z-80 block instructions. Again, the transfers are either a byte at a time per block instruction (semi-automatic) or the complete block in one instruction (automatic).

The INI instruction transfers one byte of data from an I/O device controller to memory. The C register is initialized with an I/O address as in the case of an IN R,(C) instruction. As the INI is executed, the input data is stored into the memory location pointed to by the HL register. After the data is stored, the HL register pair is incremented by one and a byte count in the BC register pair is decremented by one. The Z flag is set if the contents of the BC register pair equals zero after instruction execution. The following code shows the setup of the registers for INI execution and the INI input loop.

```
        LD   HL,BUFFER        BUFFER ADDRESS
        LD   BC,25600+23H     100 BYTES & DEV ADD
STATUS  IN   D,(C)            GET STATUS
        BIT  0,D              TEST
        JP   Z,STATUS         LOOP IF NOT READY
        INI                   INPUT ONE DATA BYTE
        JP   NZ,STATUS        LOOP IF NOT 100 BYTES
DONE    RET                   RETURN
```

The LD BC loads the B register with 100_{10} and the C register with 23H. The INI and JP instructions total 6.5 μs and as a result, the maximum data transfer speed is 50% greater than the I/O loops above. The INI replaces the following code in a conventional microprocessor.

```
STATUS  IN   A,(C)      INPUT ONE BYTE
        LD   (HL),A     STORE IN MEMORY
        INC  HL         BUMP POINTER
```

```
        DCR   C              DECREMENT COUNT
        JP    NZ,STATUS      LOOP IF NOT 100 BYTES
        RET                  RETURN
```

Although the INI is extremely convenient, as it handles the overhead operations of storage and memory and pointer maintenance, the I/O operation is still a *programmed* I/O loop. No other code can be executed until the I/O transfer is complete; the CPU is I/O bound at the speed of the I/O device. If an I/O block transfer is done to a Teletype ASR-33 printer, the transmission rate will still be 10 bytes per second, or 100 ms between bytes with most of the time devoted to the "test status" portion of the I/O loop.

The INIR instruction is identical to the INI instruction except that the total number of bytes specified in the B register are transferred with the INIR instruction. As data is input, it is stored into memory in forward (low memory to high memory) fashion. Each iteration of the INIR takes 5.25 μs if more data is to be transferred (B register not zero). The maximum data transfer rate is therefore close to 190,000 bytes per second. The INIR has no capability to test device controller status; there is no "built-in" *handshaking* logic. The question arises, then, of how the CPU is informed when the next data byte is available. The answer is that it *is not* informed. The I/O device controller must be fast enough to transfer data at a 200 kb/sec rate or must insert I/O wait states to effectively make the INIR instruction time equal to the data transmission rate of the I/O device (see Chapter 8). The actual time of the INIR instruction is, therefore, dependent on the I/O device and the INIR transfer will complete after N bytes have been transferred at the I/O device speed. If the hardware wait state is used it has been substituted for the software status check of the INI instruction. The following code shows a typical INIR input loop.

```
READC   LD    HL,BUFFER      BUFFER ADDRESS
        LD    B,200          BYTE COUNT
        LD    C,30H          I/O DEVICE ADDRESS
        INIR                 INPUT 200 BYTES
DONE    ~                    DONE HERE
        ~
        ~
        ↓
```

The IND and INDR instructions operate exactly the same as the INI and INIR instructions except that the transfers build I/O data down from high to low memory. Everything else is the same as shown next.

```
READC   LD    HL,BUFEND      END OF BUFFER
        LD    BC,25600+23H   100 BYTES & DEV ADD
STATUS  IN    D,(C)          GET STATUS
```

```
        BIT   0,D          TEST
        JP    Z,STATUS     LOOP IF NOT READY
        IND                INPUT ONE DATA BYTE
        JP    NZ,STATUS    LOOP IF NOT 100 BYTES
DONE    RET                RETURN
```

The OUTI instruction transfers one byte of data from a memory location pointed to by the HL register pair to the device whose device address is in the C register. After the transfer, the contents of the HL register pair is incremented by one and a byte count in the R register is decremented by one. The Z flag is set if the contents of the B register after the decrement is zero. The OUTI is very similar to the INI except, of course, for the direction of the I/O transfer. The following code shows the code to set up and implement the OUTI transfer.

```
        LD    HL,BUFFER        BUFFER ADDRESS
        LD    BC,25600+23H     100 BYTES & DEV ADD
STATUS  IN    D,(C)            GET STATUS
        BIT   0,D              TEST
        JP    Z,STATUS         LOOP IF NOT READY
        OUTI                   OUTPUT 1 DATA BYTE
        JP    NZ,STATUS        LOOP IF NOT 100 BYTES
DONE    RET                    RETURN
```

The OTIR is an automatic output instruction analogous to the INIR. From 1 to 256 bytes will be output from the specified block in memory according to the byte count in the C register. The same implementation as the INIR applies. If the device is slow compared to the 190,000 byte-per-second rate of the OTIR, I/O, wait states must be employed to match the speed of the I/O device and the CPU. A typical output loop using the OTIR is shown next.

```
WRITE   LD    HL,BUFFER    BUFFER ADDRESS
        LD    B,200        BYTE COUNT
        LD    C,30H        I/O DEVICE ADDRESS
        OTIR               OUTPUT 200 BYTES
DONE    )                  DONE HERE
        (
        ↓
```

The OUTD and OTDR are almost identical to the OUTI and OTIR except that the data is written from the output buffer starting at the buffer end and working back. All other actions are the same.

SOFTWARE I/O DRIVERS

The above discussion included examples of short subroutines intended to read or write one character or block of characters to an I/O device. The I/O device in question was assumed to be a rela-

tively simple device that supplied one interface signal, busy or ready. Although there are a number of devices that interface in such a simple fashion, there are many other devices that require more elaborate interfacing. A reel-to-reel magnetic tape has a variety of functions that may be performed including reading a *record*, writing a record, writing an *end-of-file*, forward or backward skips over records or files, and so forth. In addition, the tape provides many status outputs such as *end-of-tape*, *load-point*, parity error, *end-of-file*, and *off-line*. Obviously, a software routine for this device has to be considerably more complicated than just the simple loops described. To provide a complete subroutine for servicing the more sophisticated I/O devices or to *provide additional capability for the simple I/O devices*, most large software systems include a software *I/O driver* subroutine for each kind of I/O device in the system. The I/O driver handles all communication with the device type and acts as a software interface between portions of the system programs that require I/O service and the I/O devices.

To illustrate this concept, assume that several floppy discs are connected to the Z-80 system. The physical characteristics of each disc are shown in Fig. 15-1. The basic functions that one would want to perform with a floppy might be the following:

1. Read track N starting at sector M for J bytes into a specified buffer.
2. Write from a specified buffer J bytes starting from track N, sector M.
3. Position head to track N (in preparation for read or write).

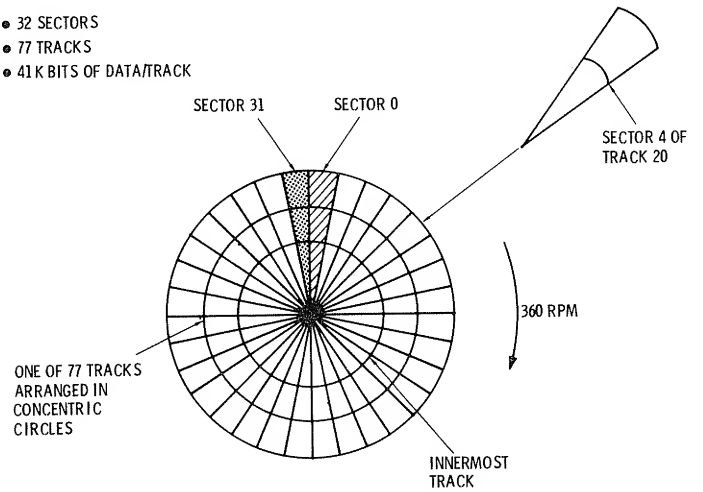


Fig. 15-1. Typical floppy-disc characteristics.

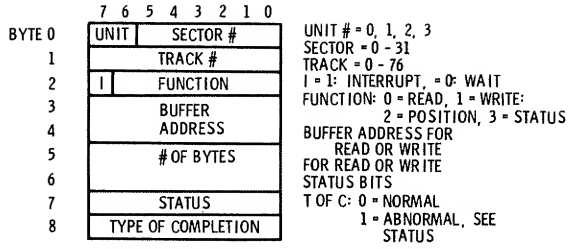


Fig. 15-2. Floppy-disc I/O driver-parameter block.

4. Read current status from the disc.

As many as four disc drives usually connect to one controller, so all of the preceding functions would also have a *unit number* of 0 through 3. Besides the previous physical functions, there may be system options that might be desirable to implement. Some of these might be:

1. Provide error indication if a read or write error.
2. Provide interrupt capability if desired.

A representative block of parameters to be passed to the floppy disc I/O driver might appear as in Fig. 15-2. Basically, four functions are implemented in word two bits 6 – 0. Function 00H is write, 01H is read, 02H is position, and 03H is read status.

For a write or read, the buffer address is held in bytes 3 and 4 and the number of bytes to be transferred is in bytes 5 and 6. The starting track and sector are in bytes 1 and 0 with the drive number, 0 – 3, in bits 7 and 6 of byte 0. The status after the read or write is loaded after the read or write operation. The type of completion is a 0, if no error occurred, or a –1, if an error occurred with further status available to define the error in byte 7.

For a position function (02H) only the number of the drive, sector, track, and function need be specified. The I/O driver would position the drive to the indicated track and sector and provide the type of complete and status.

A status function (03H) would require only the drive number and function. The I/O driver would pass the current status into the status byte and also provide a type completion code.

The I bit in byte 2 would enable a calling program to perform a "wait-for-complete" I/O if I = 0 or an *interrupt driven* I/O for I = 1. If a wait for complete I/O was required, the I/O driver would not return to the calling program until the function requested was completely finished. If I = 1, then the I/O driver would set up the proper interrupt logic in the controller, start the read, write, or position, and then return to the calling program. At the termination of

the function, the floppy-disc interface would detect the termination of the read, or write, and provide an interrupt to the Z-80, initiating the expected interrupt action.

The parameters above could be passed to the floppy-disc I/O driver in a variety of ways, but the most typical would probably be to pass a pointer in the HL or index registers.

```
        LD      IY,IOBLK        LOAD IOBLK POINTER
        CALL    DISCD           READ DISC
        LD      A,(IY+8)        GET TYPE OF COMPLETE
        BIT     7,A             TEST TOC
        JP      NZ,ERROR        GO IF ERROR ON READ
                ⟨
IOBLK   EQU     $
SECTOR  DEFS    1
TRACK   DEFS    1
FUNC    DEFS    1
BUFFER  DEFS    2
NOBYTS  DEFS    2
STATUS  DEFS    1
TOFC    DEFS    1
```

The preceding code shows a pointer to the I/O block being passed in the IY index register and an IOBLK with the location of the parameters defined. In many cases, the proper parameters will be put in dynamically at run-time rather than being preassembled, as many of them change depending on the type of I/O call.

The implementation of the I/O driver "DISCD" is not shown because it is strictly dependent on the type of floppy-disc drive and the interface design. There will be quite a bit of latitude in how the actual interface is implemented. Perhaps the 8080-type interrupt mode 0 will be implemented instead of the mode 2 table-vectored interrupts. Possibly I/O will be nonDMA instead of direct transfers to memory. In any event, the I/O driver must translate the parameters provided into proper I/O instructions to initiate and complete the I/O actions required.

In many microcomputers, firmware in the device controller performs the I/O driver functions, providing the user a more simple interface than that described.

DMA ACTIONS

In the preceding interface, DMA, or direct-memory access, may be required. In many cases, the performance of DMA will be the responsibility of the hardware logic of the device controller. It will automatically communicate with the CPU via the $\overline{\text{BUSRQ}}$ and

$\overline{\text{BUSAR}}$ signals described in Chapter 8 to "lock out" the CPU during a *cycle-stealing* access to memory to transfer data independently of CPU instruction execution. Although the format of the commands sent to the interface vary from system to system, a typical sequence would involve sending a buffer address, a byte count, and a command code to define the nature of the DMA (read, or write, and possibly other auxiliary functions). A sequence might go somewhat as follows:

```
        LD    C,30H       ADDRESS OF DISC BFR PORT
        LD    A,COMAND    START COMMAND
        OUT   (C),A       START DMA SETUP
        LD    DE,BYTES    # OF BYTES FOR DMA
        LD    BC,BUFFER   BUFFER ADDRESS
        OUT   (C),B
        OUT   (C),C       OUTPUT TO DISC BFR REG
        OUT   (C),D
        OUT   (C),E       OUTPUT TO DISC BYTE REG
LOOP    IN    A,(31H)     GET STATUS OF DMA
        BIT   0,A         TEST BUSY
        JP    Z,LOOP      GO IF STILL BUSY
```

In the preceding sequence (and it is hypothetical although representative), the buffer address and byte count are sent out in a 4-byte predefined sequence to I/O port 30H. I/O port 30H is the address of the floppy-disc controller DMA control registers. When the controller receives the "start DMA setup" command, it expects that the next 4-bytes output to port 30H will be the two bytes defining the address for the DMA and the two bytes defining the number of bytes to transfer. After receiving the fourth byte, DMA action starts immediately. Here, a second port address, 31H, may be queried for status on the state of the DMA. The code shown here is the familiar wait loop that continually checks ready status, during time of DMA activity. Since one associates DMA with automatic transfers transparent to the user program, the code at this point would probably *not* wait for DMA complete, but finish any clean-up actions required in the DISCD driver, and return to the calling program. The calling program could then perform additional processing functions until the interrupt associated with the floppy-disc DMA occurred, if an interrupt was specified, or conceivably could periodically check DMA complete by another type of call to the DISCD driver program.

INTERRUPT OPERATIONS

The interrupt operations associated with the Z-80 system operation are somewhat like the detailed I/O operations—rather difficult to describe in the general case, without reference to a particular system. The actual CPU instructions involved are the IM 0, IM 1, IM 2, EI, and DI instructions which were discussed in earlier chapters in conjunction with various interrupt modes. IM 0, IM 1, and IM 2 set the appropriate interrupt mode, which is very much related to how the system is configured for interrupts. Each device controller or PIO will have its own address and hard-wired interrupt response and probably only one of the three modes will be used for any given system. DI and EI simply disable or enable interrupts, allowing *maskable* external interrupts to become active. When the interrupt enable flip-flop is set, interrupts cause an interrupt action dependent on the interrupt mode in force (see Chapter 7). Ultimately, any interrupt transfers control to an interrupt processing routine, usually unique for every kind of interrupt.

In the interrupt processing routine, the first order of business is to ascertain that other interrupts are disabled until the contents of all registers and flags that will be used in the interrupt routine can be saved. All maskable interrupts are disabled on an interrupt until the issuance of the next EI instruction, so the following sequence can be executed:

```
PTAPE           EQU  $       PTAPE INTERRUPT ROUTINE
                PUSH AF      SAVE ALL
NO MASKABLE     PUSH BC      REGISTERS
INTERRUPT       PUSH DE        AND FLAGS
CAN OCCUR       PUSH HL
IN THIS         PUSH IX
SEQUENCE        PUSH IY
                EI           ENABLE INTERRUPTS
                  ~
                  ↓
```

Here all registers except SP have been saved, indicating that the interrupt processing will be quite extensive and will probably utilize all registers. If the interrupt processing is relatively minor, and it is known that certain registers will not be used, then those registers need not be saved. It is hard to imagine an interrupt processing routine, however, in which flags would not somehow be affected, and the PUSH AF (or EX AF,AF') must always be executed. Once the environment has been saved, then higher priority interrupts are free to occur and be processed.

Interrupt processing actions are virtually unlimited, except that

in good system design interrupt processing is kept relatively short compared to other tasks in the system. At the end of interrupt processing for the particular interrupt involved, the SP will point to the last register pair saved and the environment may be restored by pops (or exchanges) and a return made to the interrupted location by either an RETI (return from maskable interrupt) or RETN (return from nonmaskable interrupt).

```
                 RETURN    DI          DISABLE INTERRUPTS
                           POP IY      RESTORE
NO MASK-                   POP IX        ENVIRONMENT
ABLE                       POP HL
INTERRUPT                  POP DE
CAN OCCUR                  POP BC
IN THIS                    POP AF
SEQUENCE                   EI          ENABLE INTERRUPTS
                           RETI        RETURN FROM INTERRUPT
```

Both the DI and EI prevent another maskable interrupt from occurring during their execution. The EI instruction prevents an interrupt from occurring until one instruction after the EI, allowing successful completion of the RETI instruction to effect the return from interrupt.

As the maskable interrupt (NMI) can occur at any time, it may occur at a time when maskable interrupts are disabled and registers are being saved or restored or lock-out conditions for reentrancy are in force. Order may still be retained, however, if the NMI interrupt routine avoids stack use and utilizes a separate memory area to save the environment. After processing is done and an *RETN* is executed, the previous state of the maskable interrupt is restored and processing can continue. As the NMI is usually implemented for catastrophic conditions, the validity of the system may be in question at the end of the NMI routine and the recovery action may not be required anyway.

CHAPTER 16

Z-80 Programming—Commonly Used Subroutines

This chapter discusses commonly used subroutines and provides some guidance about how to efficiently implement them in Z-80 assembly language. Among the subroutines considered are comparison routines, timing loops, multiply and divide subroutines, multiple-precision arithmetic routines, routines to convert from ASCII to binary, decimal and hexadecimal, and back again, a routine to "fill" data, a string comparison, and a table search. While these are certainly not all of the routines that will ever be used in Z-80 programs, they do represent functions that need to be implemented again and again, and could be made part of a permanent *library* of routines that may be called on as required. Each subroutine discussed here is considered from the standpoint of a separate functional *module,* rather than in-line code. System design may utilize precanned modules such as these with overall system functional requirements in a combination of "top-down" (system requirements) and "bottom-up" (functional low-level subroutines) implementation.

COMPARISON SUBROUTINE

Many decisions are made based on the results of comparisons of one operand with another. A convenient subroutine for comparison would compare algebraically 8-bit operand A with 8-bit operand B and provide an indication of A<B, A<=B, A=B, A>=B, or A>B. If the call to the subroutine is followed by three relative jumps, then the calling sequence could be:

```
LD      A,(OPERA)   OPERAND A
LD      B,(OPERB)   OPERAND B
CALL    CMPARE      A:B
JR      LTHAN       RETURN HERE IF A < B
JR      EQUAL       RETURN HERE IF A = B
JR      GTHAN       RETURN HERE IF A > B
```

As in some high-level languages, the missing equalities of A<=B and A>=B may be constructed by proper use of the return points, such as this sequence that returns to location LTEQL if A<=B and to GTTHAN if A>B.

```
LD      A,(OPERA)   OPERAND A
LD      B,(OPERB)   OPERAND B
CALL    CMPARE      A:B
JR      LTEQL       A < B RTN
JR      LTEQL       A = B RTN
JR      GTHAN       A > B RTN
```

The actual code within the subroutine would be an expansion of that in Chapter 11.

```
CMPARE  CP      B           A:B
        EX      (SP),HL     GET RETURN
        JR      Z,EQUAL     GO IF A = B
        PUSH    AF          SAVE A, FLAGS
        XOR     B
        JP      P,SAME      GO IF SIGNS EQUAL
        POP     AF          RESTORE A, FLAGS
TEST    JP      C,LESST     A < B
        JP      GREAT       A > B
SAME    POP     AF
        CCF
        JP      TEST
GREAT   INC     HL          BUMP TO CALL + 4
        INC     HL
EQUAL   INC     HL          BUMP TO CALL + 2
        INC     HL
LESST   EX      (SP),HL     RESTORE RTN
        RET                 RETURN
```

The comparison is made after first retrieving the return address from the stack (stack pointer remains unaffected). The return address is then incremented to a return point to +2 or +4 bytes past the return location stored by the call to reflect the return on <, =, or >.

TIMING LOOP

A timing loop may be used for a variable time delay for I/O processing, real-time interval timing, or a delay for operator response. It is convenient to have a timing loop that will run from milliseconds to 30 seconds or so. If the average instruction execution time is 2.5 μs (4-Mhz clock), then 12,000,000 instructions will have to be executed for a 30-second delay. A two-level nested loop will work if each segment of the loop can delay about 3,000 counts or if one can delay 300 counts, while the other delays 30,000 counts.

```
DELAY   LD      DE,—1       FOR DECREMENT
LOOP1   LD      HL,14814    INITIALIZE INNER LOOP
LOOP2   ADD     HL,DE       HL — 1
        JR      C,LOOP2     GO IF NOT 0
        DJNZ    LOOP1
        RET
```

The inner loop above takes 0.1 second to decrement HL from 14814 to 0. The outer loop is determined by the B register input value so that:

$$\text{DELAY} = (\text{B}) \times 0.1 \text{ sec. approximately}$$

The maximum delay is 25.6 seconds with B = 256, but longer delays can be implemented by altering the initialization value of HL.

MULTIPLY AND DIVIDE SUBROUTINES

An 8-bit by 8-bit unsigned multiply was discussed in Chapter 12. The operands in this multiply are somewhat limited for many applications. A 16-bit by 16-bit multiply would handle most microcomputer applications; if more precision were required, a *floating point* multiply would have to be implemented. In the 16-bit by 16-bit multiply below, the multiplier is input in DE, unsigned, and the multiplicand is in BC, unsigned. The 4-byte output is passed in D, E, H, and L. Overflow is not possible.

```
MULT16  LD      A,16        ITERATION COUNTER
        LD      HL,0        ZERO PRODUCT
LOOP    BIT     7,D         TEST NEXT M'IER BIT
        JP      Z,JUMP1     GO IF 0
        ADD     HL,BC       ADD M'CAND
        JP      NC,JUMP1    GO IF NO CARRY
        INC     DE          CARRY TO MS BYTES
JUMP1   DEC     A           DECREMENT ITERATION COUNT
        RET     Z           RETURN
        EX      DE,HL
```

```
        ADD     HL,HL       SHIFT DE
        EX      DE,HL
        ADD     HL,HL       SHIFT HL
        JP      NC,LOOP     GO IF NO CARRY
        INC     DE          CARRY TO MS BYTES
        JP      LOOP        CONTINUE
```

The above implements a 16-bit by 16-bit unsigned multiply to give a 32-bit product. A *signed* multiply would require a product sign check, followed by a step to take the absolute value of the operands. The unsigned multiply could then be called and the product reconverted to the proper sign. The sign check is easily implemented by an exclusive OR of the signs of the two operands.

```
SIGNC   LD      A,D         OPERAND 1
        XOR     B           XOR OF SIGN
        LD      A,0
        JP      P,JUMP2     GO IF + RESULT
        CPL                 -1 TO A
JUMP2   LD      (SIGN),A    STORE 0 FOR +, -1 FOR -
```

A 16-bit by 8-bit unsigned divide was implemented in Chapter 12. As was the case of the 8 by 8 multiply, the operands are somewhat too small to be practical in many cases. A 32-bit by 16-bit unsigned divide is shown next that will cover divides in which the quotient can be resolved in 16 bits. The 32-bit dividend is input in H,L,D, and E while the 16-bit divisor is in register pair BC.

```
DVDE16  LD      A,16        ITERATION COUNTER
LOOP    ADD     HL,HL       SHIFT HL LEFT
        EX      DE,HL       DE TO HL FOR SHIFT
        ADD     HL,HL       SHIFT (DE)
        EX      DE,HL
        JP      NC,JUMP1    GO IF NO CARRY
        INC     HL          CARRY TO MS 2 BYTES
JUMP1   OR      A           RESET CARRY
        SBC     HL,BC       SUBTRACT DIVISOR
        INC     DE          SET Q = 1
        JP      P,JUMP2     GO IF Q = 1
        ADD     HL,BC       RESTORE
        RES     0,E         SET Q = 0
JUMP2   DEC     A           DECREMENT COUNT
        JP      NZ,LOOP     GO IF NOT DONE
        RET                 RETURN
```

At the completion of the divide routine, the 16-bit quotient is held in DE and any remainder is in HL. Overflow will occur if the quotient cannot be held in 16 bits. (No overflow indication is provided.)

A signed divide can be implemented in the same manner as the signed multiply. The sign of the quotient is determined by an exclusive OR of the signs of the two operands. The absolute values of the two operands are then taken and the unsigned divide is called. After the divide, the quotient and remainder are converted to the proper sign.

MULTIPLE-PRECISION ARITHMETIC ROUTINES

The routines in this group provide n-precision adds and subtracts. From 1 byte to many bytes of precision may be used. The add multiple routine adds the n-precision destination operand starting at the location pointed to by HL to the n-precision source operand starting at the location pointed to by DE. The n-precision result is put into the destination locations. The number of bytes of precision is contained in C. The carry is set on return to reflect the last (most significant) add.

```
MPADD   LD    B,0       NOW HAVE N IN BC
        ADD   HL,BC     POINT TO LS BYTE + 1
        DEC   HL        POINT TO LS BYTE
        EX    DE,HL
        ADD   HL,BC     LS BYTE + 1
        DEC   HL        LS BYTE
        EX    DE,HL     SOURCE POINTER IN DE
        OR    A         CLEAR CARRY
LOOP    LD    A,(DE)    GET BYTE
        ADC   A,(HL)    SOURCE + DEST + CARRY
        LD    (HL),A    STORE RESULT
        DEC   HL        DECREMENT DEST PNTR
        DEC   DE        DECREMENT SOURCE PNTR
        DEC   C
        JR    NZ,LOOP   GO IF NOT N ADDS
        RET             RETURN
```

The subtract multiple routine operates identically to the add multiple routine above, except that the destination operand is *subtracted* from the source operand and the result is stored in the destination location. The carry is set on return to reflect the last (most significant) borrow.

```
MPSUB   LD    B,0       NOW HAVE N IN BC
        ADD   HL,BC     POINT TO DEST LS BYTE + 1
        DEC   HL        POINT TO LS BYTE
        EX    DE,HL
        ADD   HL,BC     POINT TO SOURCE LS BYTE + 1
        DEC   HL        LS BYTE
        EX    DE,HL     SOURCE PNTR IN DE
        OR    A         CLEAR BORROW
```

```
LOOP    LD      A,(DE)      GET SOURCE BYTE
        SBC     A,(HL)      SOURCE-DEST-CARRY
        LD      (HL),A      STORE RESULT
        DEC     HL          DECREMENT DEST PNTR
        DEC     DE          DECREMENT SOURCE PNTR
        DEC     C
        JR      NZ,LOOP     GO IF NOT N SUBTRACTS
        RET                 RETURN
```

ASCII TO BASE X CONVERSIONS

It is convenient to have subroutines that will take a string of ASCII characters representing ASCII binary, decimal, hexadecimal, or bcd digits and convert the ASCII characters to the proper number of base representation. By employing these routines, data can be entered from a keyboard and converted to variables to be used in execution of a program, or to instructions.

The general philosophy in the following routines will be that there is a string of ASCII characters, starting at location (HL), that must be converted to the proper internal format. ASCII characters representing binary numbers will be converted eight at a time since eight bits can be held nicely in an 8-bit register. ASCII characters representing hexadecimal will be converted two at a time as two hex digits may be held in eight bits; ASCII bcd characters will be converted two at a time for the same reason. When the data represents ASCII decimal digits, the question becomes one of a convenient length for conversion. Eight bits will hold the decimal values of 0 to 255, and sixteen bits will hold up to 65,536. Neither range lends itself to consecutive conversions of strings of data as in the binary, hexadecimal, and bcd cases. The decimal conversion will arbitrarily be made of six ASCII characters, as this represents a reasonable range of values to be input to a program. After each conversion, the pointer to the string will be incremented by the number of characters converted so that subsequent conversions can be made from the next character in the string.

The following routine converts eight ASCII characters, assumed to be ASCII ones or zeros, to an 8-bit binary value in the A register. On input, HL points to the first character, and on output, HL points to the ninth character in the string.

```
ASXBIN  LD      B,8         SET COUNT
        LD      C,0         CLEAR RESULT
LOOP    SLA     C           SHIFT RESULT
        LD      A,(HL)      GET ASCII
        INC     HL          BUMP POINTER
        SUB     30H         CONVERT TO 0 OR 1
```

```
        OR      A,C         MERGE WITH RESULT
        LD      C,A         SAVE RESULT IN C
        DJNZ    LOOP        GO IF NOT DONE
        RET                 RETURN
```

The next subroutine converts two ASCII characters, assumed to be ASCII hex digits (0-9, A-F) to an 8-bit value in the A register. On input, HL points to the first character in the string, on output, HL points to the third character in the string.

```
ASXHEX  LD      C,0         CLEAR RESULT
        LD      A,(HL)      GET FIRST CHARACTER
        CALL    CVERT       CONVERT
        INC     HL
        LD      A,(HL)      GET SECOND CHARACTER
        CALL    CVERT       CONVERT
        INC     HL          POINT TO THIRD CHAR
        RET                 RETURN
CVERT   SLA     C           ALIGN RESULT
        SLA     C
        SLA     C
        SLA     C
        SUB     30H         CONVERT TO 0-15
        CP      A,10        CHECK FOR A-F
        JP      M,JUMP1     GO IF 0-9
        SUB     A,7         CONVERT A-F TO 0-15
JUMP1   ADD     A,C         MERGE RESULT
        RET                 RETURN
```

The bcd conversion routine in this set converts two ASCII characters, assumed to be the ASCII bcd digits 0-9, to an 8-bit value in the A register representing two bcd digits. As before, HL points to the first character of the string on input and the third character of the string on output.

```
ASXBCD  LD      A,(HL)      GET FIRST CHARACTER
        INC     HL          BUMP POINTER
        SUB     A,30H       CONVERT TO BCD
        RLCA
        RLCA
        RLCA
        RLCA                ALIGN TO BITS 7-4
        LD      C,A         SAVE RESULT
        LD      A,(HL)      GET SECOND CHARACTER
        SUB     A,30H       CONVERT TO BCD
        AND     A,C         MERGE
        INC     HL          BUMP POINTER
        RET                 RETURN
```

The decimal conversion routine makes use of a multiply by 10 by shifting and adding. The five characters to be converted (leading

zeros are not ignored) are pointed to by IX on entry. On exit, IX points to the start of the string + 5, and HL holds the converted result. The ASCII characters are assumed to be the ASCII decimal characters 0 through 9.

```
ASXDEC  LD    B,5     SET UP COUNTER
        LD    HL,0    CLEAR RESULT
LOOP    ADD   HL,HL   RESULT × 2
        PUSH  HL
        ADD   HL,HL   RESULT × 4
        ADD   HL,HL   RESULT × 8
        POP   DE
        ADD   HL,DE   RESULT × 10
        LD    A,(IX)  GET NEXT ASCII CHARACTER
        SUB   A,30H   CONVERT TO 0-9
        LD    E,A
        LD    D,0     0-9 IN B,C
        ADD   HL,DE   MERGE IN RESULT
        INC   IX      BUMP POINT
        DJNZ  LOOP    GO IF NOT 6
        RET           RETURN
```

Fig. 16-1 shows the actions of the four routines in the ASCII to base X conversion group.

BASE X TO ASCII CONVERSIONS

The conversions in this group operate in reverse from the ASCII to base conversions of the previous subroutines. Here, a binary value is converted to ASCII binary, hexadecimal, bcd, or decimal characters. On entry, a pointer to the buffer area is used to store the ASCII result; on exit, the pointer points to the next available storage byte in the buffer.

The next routine converts an 8-bit binary value in the C register to a string of eight ASCII binary characters stored in buffer through buffer + 7. On entry, HL points to the buffer area and on exit HL points to buffer + 8.

```
BXASB   LD    B,8     SET COUNT
LOOP    LD    A,30H   ASCII ZERO
        BIT   7,C     TEST MSB
        JP    Z,JUMP1 GO IF 0
        INC   A       ASCII ONE
JUMP1   LD    (HL),A  STORE ASCII CHARACTER
        SLA   C       SHIFT FOR NEXT COMPARISON
        INC   HL      POINT TO NEXT POSITION
        DJNZ  LOOP    GO IF NOT DONE
        RET           RETURN
```

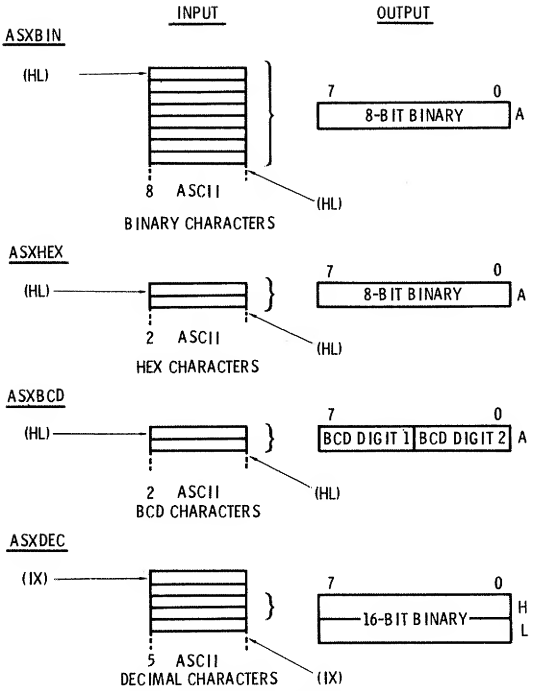


Fig. 16-1. ASCII to base X conversions.

The second routine of this group converts an 8-bit binary value in the C register to two ASCII hexadecimal digits. On entry, HL points to the buffer area and on exit, HL points to buffer + 2.

```
BXASH   LD      A,F0H       MASK
        AND     A,C         GET FIRST CHARACTER
        RRCA
        RRCA
        RRCA
        RRCA                ALIGN FOR CONVERT
        CALL    CVERT       CONVERT
        LD      A,FH        MASK
        AND     A,C         GET SECOND CHARACTER
        CALL    CVERT       CONVERT
        RET                 RETURN
CVERT   ADD     A,30H       CONVERT TO 0-15
        CP      A,10
        JP      M,JUMP1     GO IF 0-9
        ADD     A,7         CONVERT 10-15 TO A-F
JUMP1   LD      (HL),A      STORE IN BUFFER
        INC     HL          BUMP POINTER
        RET                 RETURN
```

The next routine in this set converts an 8-bit value in the C register, assumed to be two bcd digits, to two ASCII decimal digits 0-9. On entry, the HL register pair points to the buffer area and on exit, HL points to buffer + 2.

```
BXBCD   LD      A,F0H       MASK
        AND     A,C         GET FIRST BCD DIGIT
        RRCA
        RRCA
        RRCA
        RRCA                ALIGN FOR CONVERT
        ADD     A,30H       CONVERT TO 0-9
        LD      (HL),A      STORE
        INC     HL
        LD      A,FH
        AND     A,C         GET SECOND BCD DIGIT
        ADD     A,30H       CONVERT TO 0-9
        LD      (HL),A      STORE
        INC     HL
        RET                 RETURN
```

The last routine converts a 16-bit binary value in the HL register to five ASCII characters 0-9. On entry, IX points to the start of the character buffer; on exit, IX points to the start of the buffer + 5. To avoid a 16 by 8 divide with a remainder, use is made of a table lookup to find powers of ten. The table consists of 10^4, 10^3, 10^2, 10^1, and units, and is indexed by the current iteration count of the loop. A successive subtraction of the power of ten is performed to find the number of times each of the powers of ten will "go into" the residue. The number of times each can successfully be subtracted is the decimal digit for that power of ten.

```
BXDEC   LD      IY,P10TAB   POWER OF TEN TABLE
LOOP0   XOR     A           SET DIGIT CNT = 0
        LD      D,(IY)
        LD      E,(IY + 1)  LOAD — POWER OF TEN
LOOP1   OR      A           CLEAR CARRY
        SBC     HL,DE       SUBTRACT POWER OF 10
        JP      NC,JUMP1    GO IF DONE
        INC     A           BUMP DIGIT COUNT
        JP      LOOP1
JUMP1   ADD     HL,DE       RESTORE TO POSITIVE
        LD      (IX),A      STORE DIGIT COUNT
        INC     IX          BUMP BUFFER POINTER
        INC     IY
        INC     IY          POINT TO NEXT POWER OF 10
        CP      E,1         TEST FOR 5 DIGITS
        JP      NZ,LOOP0    OUTER LOOP
        RET                 RETURN
```

```
P10TAB  DEFW  10000
        DEFW  1000
        DEFW  100
        DEFW  10
        DEFW  1
```

Fig. 16-2 shows the actions of the four routines in the base X to ASCII conversion group.

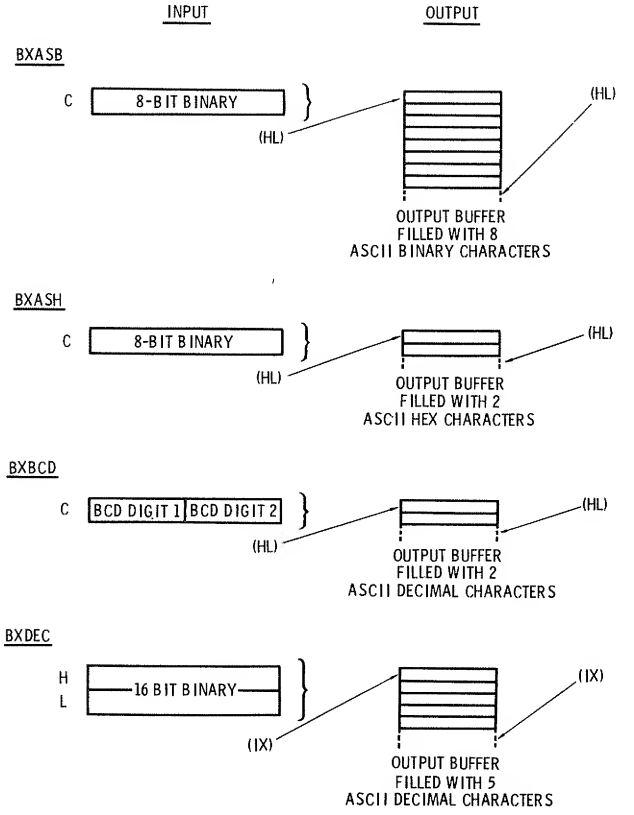


Fig. 16-2. Base X to ASCII conversions.

FILL DATA ROUTINE

The following routine fills a block of memory with a specified 8-bit binary value. This routine is useful in zeroing tables or for filling a known data value into buffers (such as all blanks to initialize a print buffer). The value to be filled is input to the A register, the starting address for the fill is contained in the HL register, and the number

of bytes to fill is in the B register. Up to 256 bytes can be filled at one time (if the B register specifies 0, then 256 bytes are filled).

```
FLDATA    LD      (HL),A      STORE BYTE
          INC     HL          BUMP POINTER
          DJNZ    FLDATA      CONTINUE IF B NOT 0
          RET                 RETURN
```

STRING COMPARISON

The following routine compares a string of data with another string of data. The strings may be of any length up to 256 bytes and typically would be search keys in a table or possibly textual strings. A comparison result indicating whether string A is less than, equal to, or greater than string B is returned. The address of string B is passed in the HL register pair and the address of string A in the DE register pair. The number of bytes in the strings (both lengths must be equal, of course) is passed in the B register. If the byte count is 0, 256 bytes will be compared. The comparison result is returned in the A register. If A<0, string A is < string B; if A=0, string A = string B; and if A > 0, string A is > string B. All comparisons are of 8-bit unsigned values.

```
COMSTR    LD      A,(DE)      GET BYTE OF STRING
          SUB     (HL)        A — B
          RET     NZ          GO IF NOT EQUAL
          INC     DE          BUMP A PNTR
          INC     HL          BUMP B PNTR
          DJNZ    COMSTR      CONTINUE IF NOT AT END
          RET                 RETURN
```

TABLE SEARCH ROUTINE

The following subroutine is a general-purpose routine for searching a table of n entries where each entry is made up of m bytes. The key for the search is eight bits and the search value for each entry is assumed to be in the first byte of the table. The parameters input to the routine are the search key in the A register, the start of the table in the IX register, the number of entries in the B register, and the number of bytes per entry in the E register. These are illustrated in Fig. 16-3. The search progresses down through the table. If the key is found, the address of the first matching entry is returned in the IX register; if the key is not found, a −1 is returned in IX. All parameters in A, B, and E are maintained so that the subroutine can be immediately reentered to search for the next matching key; the value of the search key and number of bytes per entry will be unchanged, and the number of entries in B will be modified to reflect the number

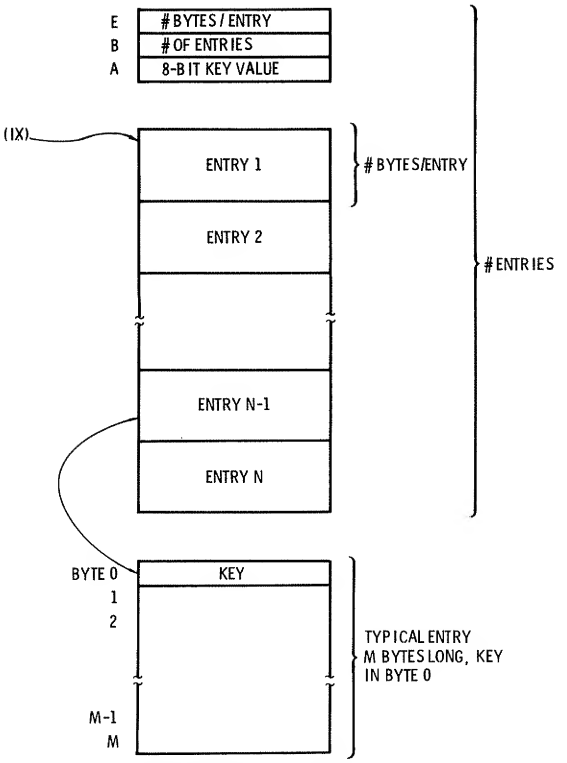


Fig. 16-3. Table search routine.

of entries remaining in the table. The reentry address is a second entry point in the routine. The initial entry is at SRTAB, while the "reentry" point is at SRTAB1.

```
SRTAB   LD      D,0       NOW HAVE # BYTES/ENTRY IN D,E
LOOP    CP      A,(IX)    COMPARE SEARCH KEY TO ENTRY
        RET     Z         GO IF MATCH
SRTAB1  ADD     IX,DE     BUMP IX TO NEXT ENTRY
        DJNZ    LOOP      GO IF NOT LAST ENTRY
        LD      IX,—1     FLAG FOR NOT FOUND
        RET               RETURN
```

SECTION III

Z-80 Microcomputers

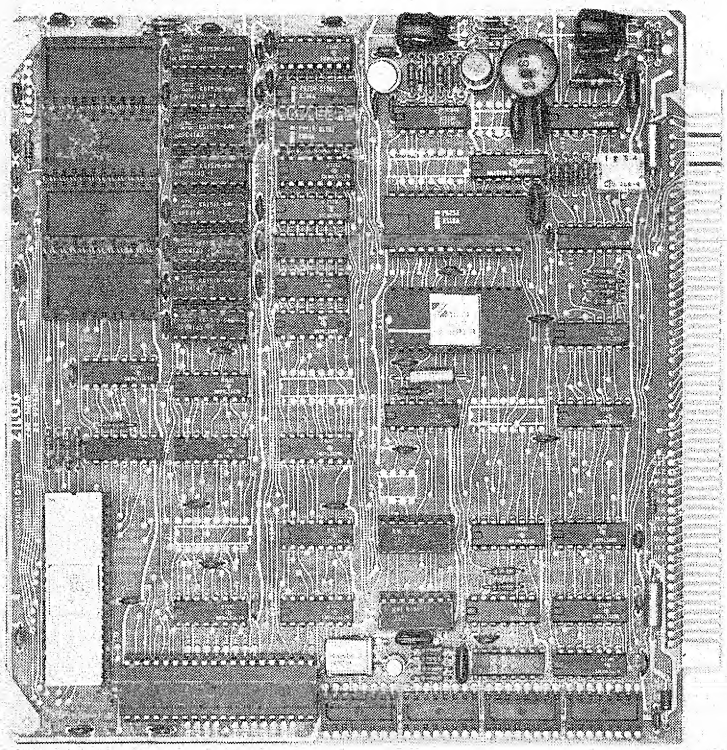
CHAPTER 17

Zilog, Inc.

Zilog, as do many microprocessor chip manufacturers, offers a complete microprocessor development system, the Z-80 Development System. This is a turn-key system with which the commercial user can develop Z-80 programs. The Z-80 Development System includes dual floppy-disc drives, 16K capacity RAM memory, full software including a debug package, operating system, editor, assembler, and file maintenance. In addition to the development system, Zilog offers a complete set of microcomputer modules including a Microcomputer Board (MCB), Disc Controller Board (MDC), and a RAM Memory Board (RMB). The modules may be supplemented by chassis, power supplies, and floppy discs to make up the microcomputer configurations desired.

Z-80 MCB™ MICROCOMPUTER BOARD

The Z-80 MCB™ is a complete 7.7- × 7.75-inch single-board computer shown in Fig. 17-1. A block diagram of the MCB is shown in Fig. 17-2. The heart of the MCB, of course, is the Z-80 microprocessor chip. RAM memory consists of 4K of dynamic RAM on the MCB board. Up to 4K of EPROM, PROM, or ROM may be used on the MCB. Zilog provides a monitor program that is available in 1/2K, 1K, and 3K versions. Parallel I/O capability is provided on the MCB with a single PIO chip. I/O capability for serial I/O devices such as teletypewriters and teletypewriter-compatible devices is implemented by a USART chip (Universal Synchronous/Asynchronous Receiver/Transmitter). A Zilog CTC™, Counter Timer Circuit, provides a real-time clock capability and is also used as a program-



Courtesy Zilog, Inc.

Fig. 17-1. Zilog Z-80 MCB™ microcomputer board.

mable baud-rate generator. A 19.6608-MHz crystal oscillator provides both the basic 2.547-MHz clock for microprocessor operation and frequencies for various serial I/O. Three-state buffers are used on all data, address, and control lines and a 122-pin connector is used to provide interfacing to other logic in the system, or compatible MCB modules. External power supply requirements are +5 VDC at 10 watts maximum.

MCB MEMORY

The nominal memory mapping for the MCB is shown in Fig. 17-3. The 4K bytes of EPROM, PROM, or ROM are normally located in block 0 (memory locations 0000H to 0FFFH) with the 4K bytes of RAM in block 1 (1000H to 1FFFH). Both the read-only memory and RAM, however, can be relocated anywhere in the 64K byte

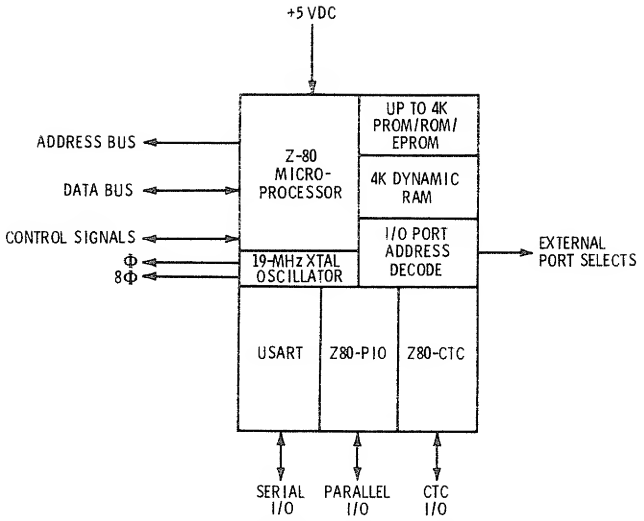


Fig. 17-2. Z-80 MCB™ functional block diagram.

addressing range of the Z-80 by slight modifications to the MCB board. The MCB uses 4K bytes of dynamic RAM in an eight-chip $4K \times 1$ configuration. At least part of the possible 4K bytes of ROM in the MCB will probably be one of the versions of the Zilog monitor. The remaining area available for read-only memory can

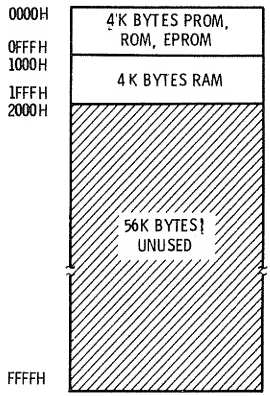


Fig. 17-3. Z-80 MCB™ nominal memory mapping.

be used for EPROM (erasable programmable read-only memory) or ROM (read-only memory). As EPROMs require +5V, −5V, and +12 VDC, these three voltages must, of course, be externally supplied to the MCB when EPROMS are used. EPROM use also requires minor modifications to the MCB.

MCB I/O PORTS

The MCB provides address decoding both for the on-board I/O sections (CTC, PIO, and serial data) and for external I/O devices. By jumping the proper pins, eight groups of 32 device addresses are selectable. The MCB is provided with the I/O address decode in the block of addresses C0H through DFH. A decode of the two least significant address bits AB1 and AB0 results in four signals (SG0, SG1, SG2, and SG3) provided externally on the connector. A decode of address lines AB7 through AB2 generates eight signals, GP0 through GP7. Three of the latter are used to address the CTC, PIO, and serial I/O section (GP5, GP6, and GP7). The remaining five can be used in conjunction with SG0, SG1, SG2, and SG3 to address external I/O devices. External I/O devices may also be addressed by an external decode of the address lines and control signals which are also brought out to the connector.

MCB PARALLEL I/O

The MCB offers a two-port programmable I/O interface by means of the PIO device on the board. The PIO enables bidirectional data transfers with full handshaking and interrupt capability as discussed in Chapter 8. The I/O ports of the PIO connect directly to a set of wire-wrap pins that can be connected to four 16-pin IC sockets on the MCB board. Appropriate drivers can be provided in the sockets for external I/O equipment and wire-wrapped connections can be easily made to edge connector pins.

MCB SERIAL I/O

Serial I/O is provided by the 8251 USART on the MCB. The 8251 is a programmable communication interface that can be programmed to handle virtually any serial data transmission scheme now in use. Baud rates from 50 to 38,400 and above are selectable by jumper connections. One of the four channels of the CTC is used to generate a baud-rate clock with the USART. The CTC is configured by the MCB software to provide a clock signal 32 times the serial communication frequency. Either an RS-232 or current-loop interface may be selected by jumper connections to provide direct connection of a teletypewriter, or many other serial data devices. With proper programming and slight circuit modifications, the USART may be configured to perform the functions of a *modem* (modulator/demodulator) to allow direct phone-line communications with the USART and MCB.

MCB INTERRUPTS

Both the PIO and CTC operate in the mode 2 interrupt mode of the Z-80, with daisy chaining between the CTC and PIO. The IEI and IEO signals for the interrupt priority chain are brought out to edge connector pins to allow external I/O devices to be connected for interrupt mode 2 servicing. The external I/O devices operate in conjunction with PIO interrupt logic as discussed in previous sections.

MCB CONFIGURATIONS

The minimum configuration for the MCB consists of the MCB, a 5-volt power supply, and a teletypewriter as shown in Fig. 17-4. The MCB must have monitor software installed in the read-only memory

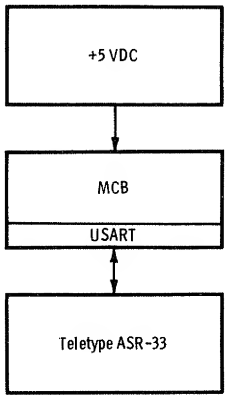


Fig. 17-4. Minimum MCB system.

section. Either the 1/2K or 1K version will suffice. This configuration offers a complete program development system as a teletypewriter offers keyboard input and output, and, if the Teletype ASR-33 is employed, paper-tape input and output. Programs may be developed and debugged with this system and saved on paper tape. Note that assembly-language capability is *not* provided; and that assembly-language programs (such as a time-sharing service) must be assembled on another system and then loaded into the MCB system. Limited debugging capability is provided by the MCB monitor.

Other modules in the MCB series provide the capability of system expansion. A Memory/Disc Controller Board (Z-80 MDC) adds the capability of communication with a floppy disc (Shugart 800) and additional RAM memory. The board provides an additional 12K bytes of RAM with complete buffering and control on the board (no additional logic is required). The disc-controller portion of this

board provides control of up to four floppy-disc drives. The board provides CRC parity generation and checking and full-status reading and control under CPU software control. An extra bidirectional 8-bit PIO port is provided with the addition of this board. The 2K and 4K versions of the MCB system monitor must be used when the Z-80 MDC board is added to provide software for the disc I/O operations. The MDC is fully compatible with the MCB board and requires only a 5-VDC supply.

A Z-80 RMB, or RAM Memory Board, is another module in this series. Additional dynamic RAM storage of 16K bytes is provided with the RMB board. The RMB board is fully compatible with the MCB and MDC boards and requires only a 5-VDC supply.

Other modules in this series include a VDB, or Video Interface Card, PROM memory boards, serial and parallel I/O boards, PROM or EPROM programmer boards, wire-wrap boards, extender boards, and a standard card cage that will hold up to nine standard modules.

A complete microcomputer system, the Z-80 MCS™, is also available from Zilog. The MCS uses the MCB and MDC and provides dual floppy discs, power supplies, card cage, and chassis.

MCB MONITOR

A description of the 1K version of the Z-80 MCB PROM™ monitor is provided below. The word monitor as applied to microcomputer systems essentially means "debugging program" in the smaller versions, as the program offers a set of debugging commands that will display the register contents; memory contents; alter register or memory with given data; save to paper tape and load memory from paper tape; and control program execution while debugging. The monitor for the MCB is discussed, not so much to provide a detailed example of this support program, but as an example of the capabilities of an on-board monitor of this kind.

The commands in the 1K MCB monitor are as shown in Table 17-1. "AAAA" stands for an address value, "NN" stands for a number

Table 17-1. MCB 1K Monitor Commands

Sequence	Command
1	DISPLAY AAAA NN CR
2	SET AAAA DD DD DD DD . . . DD CR
3	REGISTER RR (blank)
4	BREAK AAA {N} CR
5	JUMP AAAA CR
6	GO CR
7	PUNCH AAAA AAAA CR
8	LOAD CR

of bytes, "DD" represents two hexadecimal digits, and "R" is a mnemonic for a register.

The commands above are entered on the system entry device which would typically be a teletypewriter or video display keyboard. The monitor will recognize the command by its first letter, and typing the full command is optional. Each field of the command is *delimited* by a blank. All numeric values represent hexadecimal digits. Leading blanks may be omitted, and entering more than four characters results in the last four being used as the value. The (CR) represents a carriage return character on the keyboard device. If an invalid command is entered, the monitor responds with a question mark *prompt* and a new command can be input.

The DISPLAY command displays the contents of address AAAA to address AAAA + (NN − 1) on the display device. To display locations 1000H to 10FFH, the user would type:

DISPLAY 1000 100 (CR)

SET stores the given data bytes, which may be any number, into memory starting at location AAAA. To store the values 1, 2, 3, 4, etc., into the locations from 0300H, the user would type:

SET 300 1 2 3 4 5 6 7 8 9 10 (CR)

REGISTER displays the contents of a given register. The user types in the command "REGISTER" followed by a one- or two-letter mnemonic signifying the register to be displayed. Valid register mnemonics are A, B, C, D, E, F, H, L, I, A′, B′, C′, D′, E′, F′ H′, L′, IX, IY, PC, or SP. The mnemonic is followed by a blank rather than a carriage return. To display the contents of register IX, the user would type:

REGISTER IX (b)

After the mnemonic and blank have been typed, the monitor will print the contents of the register on the same line. The user then has three options. If the register is not to be modified, and the next register is not to be displayed, a carriage return may be typed. If the register is not to be modified, but the next register in the sequence A, B, etc., is to be displayed, a line feed may be entered; the next register in sequence will then be displayed. The contents of the register may be modified to a new value by typing a blank, a new value, and a carriage return or linefeed. To modify registers A′ through E′ the following sequence would be followed:

```
REGISTER A' 12 13 (LF)    (A' modified to 13 from 12)
            14 15 (LF)    (B' modified to 13 from 14)
```

15 (LF) (C' unchanged)
16 17 (LF) (D' modified to 17 from 16)
17 18 (CR) (E' modified to 18 from 17)

BREAK sets a *breakpoint* at address AAAA. A breakpoint is an instruction location which, when executed, returns control to a monitor, or debugging program. A typical debugging method is to set a breakpoint, execute the program that is being debugged from a location before the breakpoint, and then examine the contents of memory and/or registers when the breakpoint is reached (if it is reached!). The BREAK command replaces the instruction at the specified address with a restart to 38H instruction. When the instruction that is breakpointed is executed, the RST 38H is instead executed, and the breakpoint routine is entered. The contents of all registers are then saved and a message indicating that the breakpoint has been reached is printed by the monitor. If the optional N field is entered along with the address of the breakpoint, the breakpointed instruction at AAAA will execute N times before the break occurs. This is helpful in breakpointing iterative code, as it eliminates a breakpoint at each repetitions of an instruction. To set a breakpoint at location 10AAH after 123H times through a loop, the following command would be entered:

BREAK 10AA 123 (CR)

The JUMP command causes the monitor to jump to the specified address. If previous breakpoint was entered, all registers are restored to their contents before the breakpoint. The following command starts execution at location 123AH:

JUMP 123A (CR)

The GO command is similar to the JUMP command, except that it is used after a breakpoint to continue execution from the breakpointed location.

The PUNCH and LOAD commands are used to save and restore the contents of memory. Typically, this would be done after debugging a program or partially debugging a program by *patching*. PUNCHing the contents of a block of memory saves the contents of memory on paper tape; the paper tape can subsequently be reloaded by a LOAD command to restore the previous contents.

The format of the PUNCH command specifies a starting address for the punch and an ending address for the punch. To save locations 1000H through 1FFFH on paper tape, the command would be:

PUNCH 1000 1FFF (CR)

The monitor would then punch tape *leader* (blank or null tape), followed by memory locations 1000H through 1FFFH, or about 100 inches of paper tape at 10 characters (bytes) per inch. At the end of data, a *trailer* of blanks would be punched.

The punched paper tape could then be read into memory at any later time by inputting the LOAD command to the monitor. The monitor would then load the tape, inputting data on the first non-blank character of the paper tape. Data regarding the memory locations to be loaded and checksum data is stored on the tape along with the actual memory data. If the tape is loaded correctly, the monitor will await the next command; if invalid data has been read, the message "BAD CHKSUM"will be printed.

The monitor program described above offers a convenient way to debug short programs in the MCB. The disc versions offer even more debugging aids, disc I/O capability, and rudimentary file-management functions.

Z-80 DEVELOPMENT SYSTEM

The Zilog Z-80 Development System is a complete program development and hardware development system. The basic system includes a Z-80 CPU with 4K bytes of ROM, 16K bytes of RAM (expanded to 60K bytes), and two floppy-disc drives with a controller. A teletypewriter or other terminal may be connected to the system by the RS-232 or current loop serial interface included in the system. A programmable hardware breakpoint module allows *hardware* breakpointing on specified control signals and/or addressing and data bus configurations. The user may develop his system by an in-circuit emulator which essentially connects the user's hardware with the Z-80 Development System resources. An optional parallel I/O card enables interface of the system to other kinds of peripheral equipment, such as line printers, paper-tape punches, and so forth. System software includes a ROM-based operating system and debug package, and a resident assembler, editor, and file-maintenance package.

Z-80 DEVELOPMENT SYSTEM HARDWARE

The development system is made up of modules as shown in Fig. 17-5. The Processor Module contains the Z-80, 3K bytes of ROM, and 1K bytes of RAM. This memory contains the operating system, peripheral drivers, bootstrap loader, and debug software. The editor, assembler, and file-maintenance routines are stored on floppy disc and are available on command from the operating system.

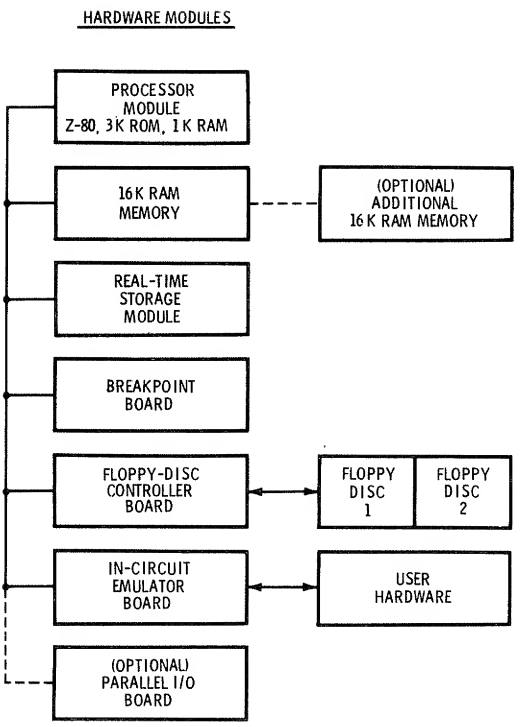


Fig. 17-5. Zilog Z-80™ development system.

System memory consists of one 16K byte memory module made up of dynamic RAM. Memory is expandable to 60K bytes of user memory (4K of addresses are utilized by the Processor Module). The additional memory below and beyond the 4K used in the processor may be in either Monitor Mode or User Mode. In User Mode, the entire set of additional memory is dedicated to the user's software; the user has control over system peripherals and CPU. In the Monitor Mode, this memory serves as main memory for editing and assembling.

The Real-Time Storage Module Board contains a 256 by 32 "storage array." The storage array essentially stores events in the array as they occur on a *rolling* basis. The user may specify the kind of events to be recorded under control of the debug software, and may specify combinations of memory reads or writes, and I/O reads or writes. As each event occurs during user program execution, the state of the address bus, data bus, and control bus are stored in the next location of the 256 places in the array. When the system returns to Monitor Mode from User Mode, the array may be interpreted and printed to enable the user to analyze the last events that occurred.

The Hardware Breakpoint Board enables the user to specify hardware breakpoints that effectively "trap" the occurrence of events that have a specified combination of address bits, data bits, and I/O bits. The Breakpoint Board, Real-Time Storage Module Board, and software debug module are used together to facilitate tracing system activity during user debugging.

The Floppy-Disc Controller Board is the hardware interface between the two floppy discs of the system and the Z-80. The optional Parallel I/O Board contains two Z-80 PIO chips which can be software configured as required to interface the system to other kinds of peripheral equipment, such as line printers, paper-tape equipment, or PROM programmers.

The In-Circuit Emulator Board is the interface between the user's own equipment and the Z-80 Development System. It is an extension of the system bus including cabling to the address bus, data bus, CPU control signals, and system clock. Additional logic controls User and Monitor Modes.

Z-80 DEVELOPMENT SYSTEM SOFTWARE

The Z-80 Development System is controlled by a small operating system, OS Z-80, which retrieves the assembler, editor, and file maintenance from disc storage. OS Z-80 is resident in the read-only memory of the Processor Module.

The debug software of the system is also resident in read-only memory and contains commands similar to the MCB debug commands. In addition, there are commands related to the Hardware Breakpoint Module and the Real-Time Storage Module; these commands specify the breakpoint and storage conditions and allow a history of events to be printed.

The Editor software is brought into memory from disc under control of OS Z-80. The Editor is a *line-oriented* editor which allows a user to append, delete, or insert lines of text and provides file management of the text files. A powerful feature of the Editor is a MACRO capability, which permits a user-specified set of commands to be invoked as required.

The Assembler of the system is a subset of the Z-80 cross assembler. The Assembler is invoked by an OS Z-80 command, which loads the Assembler software from the floppy disc. The Assembler permits assembly of text files from disc with object output also stored on disc. The object module representing the assembled file may then be loaded for execution by a Debug LOAD command. The ability to text edit, assemble, and load from floppy disc greatly facilitates program development when compared to a system operating only with paper-tape input and output.

The last item of utility software supplied with the Z-80 Development System is the File-Maintenance software. This software permits file management of text, or other files, stored on the floppy discs. Under control of the File Maintenance software, disc files may be renamed, erased, copied from one disc to another, appended to another file, or combined. File contents may be printed or punched and information about the current files, or disc, may be listed on the system printing device.

OTHER ZILOG PRODUCTS

In addition to the MCB modules and Z-80 Development System, Zilog offers a Z-80 *Simulator* program and Macro Cross Assembler for use on time-sharing services or larger computer systems. The Simulator will execute Z-80 programs by interpreting Z-80 instructions while running in a host computer. The Macro Cross Assembler allows assemblies of Z-80 programs on machines other than the Z-80. PL/Z, a higher-level language similar to PL/I, and BASIC are also offered for off-line support of Z-80 products.

CHAPTER 18

Other Z-80 Microcomputer Systems

The Zilog MCB series and Z-80 Development System discussed in the previous chapter are primarily oriented toward the commercial user of Z-80 hardware. The systems are designed to provide powerful tools in Z-80 hardware and software development. In many cases, the user will design his own specialized Z-80 System around the Z-80 microprocessor chip, using the Z-80 Development System for research and development, or for new versions of *firmware*, or software, for production systems. In other cases, modules of the MCB series will be incorporated in new designs.

The manufacturers discussed in this chapter have a somewhat different orientation. While the systems discussed here are being offered to the commercial user or OEM (Original Equipment Manufacturer), they also are being offered to the *computer hobbyist*. As a result, many of the system components are available as kits in addition to fully assembled modules. Many of the recent hobbyist kit manufacturers have produced products that have not come up to commercial standards in design, production, documentation, or support. The four manufacturers discussed here all provide quality products that are well designed, produced to commercial standards, and, in some cases, quite innovative. Other manufacturers of Z-80 equipment also are producing quality components and the discussion of these four manufacturers is not meant to imply that other units are not equal in quality.

The four manufacturers described in this chapter are Technical Design Labs, Inc.; Cromemco, Inc.; The Digital Group, Inc.; and Radio Shack. The products produced by Technical Design Labs and Cromemco are S-100 bus compatible. The S-100 bus has become a

de facto standard bus since MITS, Inc. produced the first hobbyist microcomputer, the MITS 8800, using the 8080A as a base. The signals defined for their system bus, many of which are identical or very much related to the 8080A pin signals, have become the MITS bus. As 100 pins are involved (and because other manufacturers are reluctant to promote competitors), the bus also has become known as the S-100 bus. Cromemco and Technical Design Labs products will, therefore, operate compatibly with other products that have been designed for the S-100 bus, and there are a great many such products.

The Digital Group, however, has established their own bus which is not S-100 bus compatible. There are both advantages and disadvantages to this. If a microprocessor is widely different from the 8080A-related bus, then much logic has to be devoted for conversion from the microprocessor logic signals to S-100 bus signals. Subsequent timing problems and bus status problems also have to be dealt with. On the other hand, if a module is S-100 bus compatible, then a system user may utilize all of the diverse products available for the S-100 bus in his system, ranging from speech synthesizers to special-purpose I/O interfaces. Fortunately, the Digital Group offers a wide variety of modules for their bus and this disadvantage is somewhat alleviated.

The fourth manufacturer, Radio Shack, offers a turnkey microcomputer system available to the computer hobbyist or small business user only in assembled form. The bus, of course, is not an S-100 bus, although an interface between the Radio Shack system and S-100 bus may be made available.

The following discussion is not meant to compare the four manufacturers point by point, but rather to provide a factual description of what is currently being offered by each. Since the microcomputer market is so dynamic, these descriptions will suffer with time, but should provide some guidelines to the reader in making an evaluation of Z-80 microcomputer equipment.

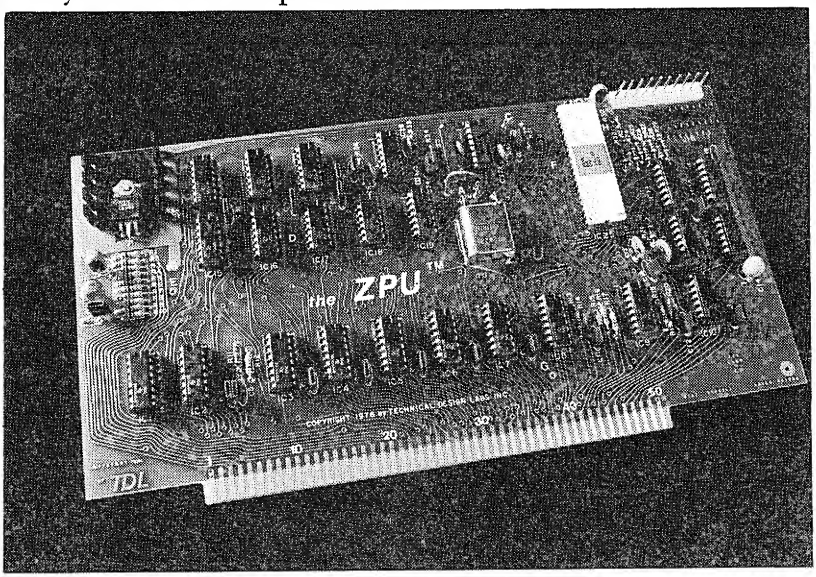
TECHNICAL DESIGN LABS, INC.

Technical Design Labs' basic Z-80 module is the TDL ZPU™ card which is an S-100 bus-compatible CPU board. In addition to the ZPU, TDL offers an S-100 16K byte memory card, and a System Monitor Board, which contains a monitor, RAM, and I/O porting. In addition to the Z-80 modules, TDL has a mainframe microcomputer, the TDL XITAN Microcomputer, which provides a system package based on the Z-80 modules. While not offering a great deal of I/O hardware at this time of writing. TDL software appears to be very impressive. TDL offers a monitor, a line and character-

oriented text editor, a *relocating macro* assembler, a BASIC and SUPER-BASIC interpreter, a text output processor, and disc software for an S-100 bus-compatible floppy disc of another manufacturer.

TDL ZPU™ BOARD

The TDL ZPU™ card is supplied either as a kit or fully assembled and tested. The board is shown in Fig. 18-1. The board contains the Z-80 microprocessor chip and buffering for the Z-80 bus signals. Most of the other logic is devoted to generation of S-100 bus-compatible signals. The ZPU™ has two clocks on the board. One is fixed at 2 MHz by a crystal oscillator; the other is frequency variable by means of a small potentiometer. The frequency of the second clock can be adjusted from about 1 MHz to 4 MHz if the second clock is selected. Either of the two clocks, or an external clock, may be selected for Z-80 timing, but the system clock (the bus clock) is always the 2-MHz output.



Courtesy Technical Design Labs, Inc.

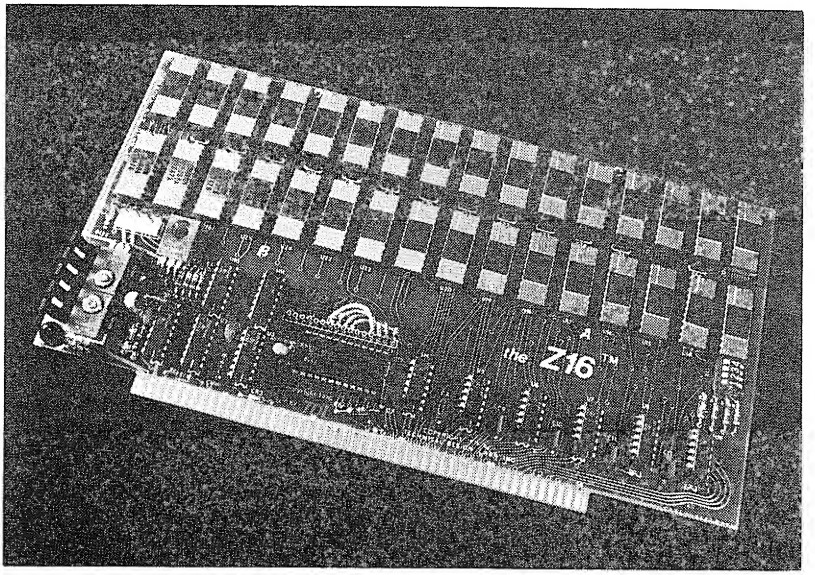
Fig. 18-1. TDL ZPU™ board.

When the ZPU™ is installed to replace an 8080A microprocessor card in an older system, there are certain options which may be utilized. The S-100 bus has no provision for a RFSH signal from the Z-80. An existing S-100 bus signal, SSTACK (pin 98), may be utilized to bus the refresh signal to the remainder of the system. The

$\overline{\text{NMI}}$ signal, nonmaskable interrupt, may be connected to the S-100 bus by connecting the $\overline{\text{NMI}}$ signal to the VIO signal of the S-100 bus, the highest-priority vectored-interrupt line on the bus. The ZPU™ board requires +5 VDC only.

TDL Z16™ BOARD

The Z16 board (Fig. 18-2) is a low-power, *static* memory board with a cycle speed of 200 nanoseconds. The board may be obtained in kit, or assembled form, with memory increments from 4K to 16K bytes. The fully populated version is 16K bytes (thirty two 4K by 1-bit chips). Power-supply voltages required for the Z16 are ±5 VDC, and 12 VDC, and are available on the S-100 bus.



Courtesy Technical Design Labs, Inc.

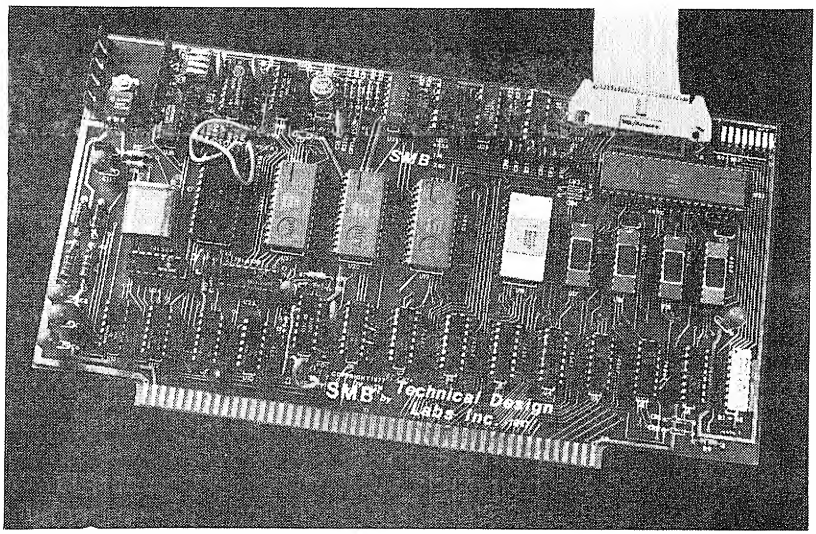
Fig. 18-2. TDL Z-16™ board.

The Z16 board has several unique features. One of these is a memory-protect capability which will prevent selected 1 to 4K segments of the 16K board from being accessed for a write. The memory protect is switch selectable. A battery backup capability is provided on the board, with the addition of an external battery pack with voltage sensing. Logic for *memory-bank* switching is also provided on the board. By means of an external Memory-Management Board, banks of 16K memory may be connected, or disconnected, from the system bus. This means that two (or more) complete 64K

banks of memory may be alternately connected and disconnected from the system bus under software control for *multiprogramming* or other uses.

TDL SYSTEM MONITOR BOARD

The SMB board is a multipurpose board that provides a monitor, RAM, and I/O device controllers. The SMB contains a 2K monitor, the ZAPPLE™ monitor in PROM. The monitor is a comprehensive debug package and collection of software I/O drivers, including a serial I/O driver and an audio tape-cassette driver. In addition to the 2K of PROM, the SMB contains 2K bytes of RAM, available to the system user. Two serial I/O ports are available. They can provide serial I/O at 110 to 9600 baud and interface to either a 20-mA current loop, or RS-232 device. Baud rates of 110 (Teletype), 300, 600, 1200, 4800, and 9600 are jumper-selectable. One parallel I/O port is available to the user and can be software configured to either an input, or output port.



Courtesy Technical Design Labs, Inc.

Fig. 18-3. TDL SMB™ board.

A second parallel I/O port (on a Motorola PIA, a programmable I/O device similar to a PIO) connects to a set of 8-dip switches which are used to configure the SMB board for I/O. By means of proper settings on the dip switches, the four system *logical devices* can be set to a *physical device*. The four system logical devices are the console device, tape-reader device, and listing device. These

logical devices may be set to various physical devices that can be connected to the system-teletypewriter, high-speed serial (CRT), paper tape, line printer, cassette, or user-defined device. As these switches are examined on system reset they allow simple reconfiguration of system devices without rewriting monitor software.

The SMB contains a built-in 1200-baud audio-cassette interface enabling data to be read from a standard Phillips-type cassette recorder. Also provided on the board is the capability to automatically restart to the monitor program on system power on and/or reset.

TDL XITAN™ MICROCOMPUTER

The ZITAN™ Microcomputer is essentially a power supply, card cage, minimal front panel, and other packaging to contain the TDL ZPU™, SMB™, and Z16™ boards. In this utilization, it is identical with most other "box without front panel" microcomputers including the Zilog Development System. If the manufacturer provides an adequate monitor, there is little reason for a front panel, as the front panel functions, and a great deal more, are implemented in the monitor.

TDL SOFTWARE

The ZAPPLE™ monitor in the SMB board provides over 20 debug and file commands. In addition, the monitor contains I/O routines for standard system devices or user-defined (user-written) I/O devices. All I/O in the other utility software is done via the I/O drivers in the ZAPPLE™ monitor.

The ZAPPLE™ text editor is both a line-oriented and character-oriented text editor that allows the user to edit text files. Lines may be inserted, or deleted. Character strings may be located by a search command and another string may be substituted, in addition to normal insertion and deletion of character strings. A macro capability for text editor command strings may also be used, to enable automatic repetition of given command strings.

A Relocating Macro Assembler is available from TDL. Both the relocatability and macro capability are sophisticated additions to the minimal assemblers seen from many microcomputer manufacturers. Relocatability allows an assembled object program to be loaded into any portion of memory. The alternative to relocatability is an absolute object output which can only be loaded and executed in one area of memory. When a great deal of system software development is to be done, relocatability facilitates changes to the programs and permits a larger program to be partitioned into relocatable modules. Changing various parts of the system does not require changing

and/or reassembly of each module as might be the case if all of the programs were absolute.

Macros are *assembly-time* predefined segments of code (as opposed to subroutines, which are run-time segments of code). Macros are made up of one, or more, instructions and constitute a section of code that performs a specific function, or set of functions. A macro may be invoked at assembly time to repeat the instructions associated with it. The net effect is to substitute a multisource line macro *expansion* for a macro call and a set of parameters defining the inputs to the macro. Calling the macro avoids writing the source lines associated with the macro at each point in the program where the function is to be performed; the macro call generates the instructions automatically. Macros also may be defined in the general case, so that a macro call may have macro parameters, or arguments defined, with the macro to generate code related to the arguments. Macro capability allows such things as assembly-time simulation of computers other than the host computer and predefined procedures.

TDL offers a BASIC and SUPER-BASIC higher-level language. Both are interpretive BASICs, that is, the source language statements or compressions of them are interpreted and executed at run-time rather than producing an executable output module at compilation time. The 8K BASIC is very complete and has such features as a trace function to display line numbers as the lines are executed and a listing of program variables. The 12K SUPER-BASIC is an expanded BASIC with many editing and formatting commands appended to the basic functions.

The Text Output Processor is a utility package which provides output formatting for text files. Output files are prepared by the Text Editor with embedded format control words. When the file is listed by the Text Output Processor, the processor performs formatting based on the control words to control line spacing, headings, page length, spacing, and other word-processing functions.

CROMEMCO, INC.

Cromemco, Inc. offers two microcomputer systems designed around their Z-80 CPU card, the Z-1 Microcomputer System, and the Z-2 Computer System. Both the Z-1 and Z-2 use S-100 bus-compatible boards in the system. The Z-1 has a control panel, while the Z-2 does not (see Figs. 18-4 and 18-5). In addition to the Z-80 CPU card, Cromemco offers a variety of other boards including 4K and 16K byte RAM, an 8K PROM board with an integral programmer, an I/O board, a PROM programmer board, a digital-to-analog I/O board, and a color graphic interface. Software includes a 1K monitor, Z-80 assembler, and CONTROL BASIC.

Courtesy Cromemco, Inc.

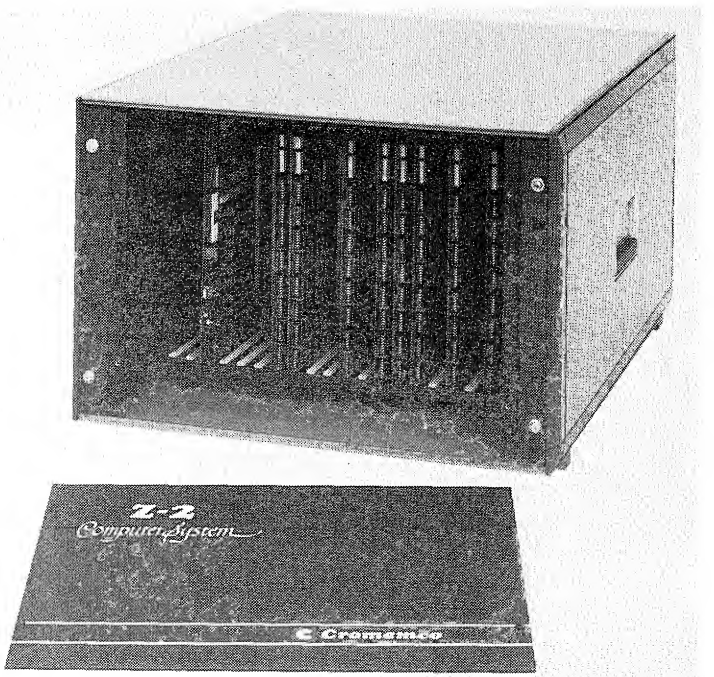
Fig. 18-4. Cromemco Z-2 computer system.

CROMEMCO CPU CARD

The Cromemco CPU board is primarily made up of the Z-80, buffers, and logic associated with S-100 bus compatibility. The clock on the CPU board is selectable to either a 2-MHz or 4-MHz clock rate. Cromemco will guarantee operation of the Z-80 microprocessor at 4 MHz, but the feature is nice as many older systems have inherent limitations of the system clock rate due to slow memories, or other system components. As in the case of TDL, the SSTACK line in the S-100 is used for a Z-80 function, this time to indicate whether the clock is set at 2 MHz or 4 MHz.

Another feature of the CPU board is an on-board WAIT-state generator. The WAIT state for the S-100 bus is identical in function to the WAIT state in the Z-80—it allows slow memory, or I/O devices, to send a WAIT signal to the CPU to provide extra time to respond to the CPU. A jumper-selectable WAIT-state option on the CPU board inserts a WAIT state in the Z-80 between the T2 and T3 cycles of every machine cycle. This enables the CPU board to be used with slower speed memories without additional WAIT logic generated by the memory board itself.

The power-on memory jump option on the CPU board allows the user to select a jump address on any 4K boundary. A jump is auto-



Courtesy Cromemco, Inc.

Fig. 18-5. Cromemco Z-2 computer system (cover removed).

matically executed to that address when the system reset switch is activated. The jump address would probably be the start of the system monitor to enable the user to load programs for execution on start-up, or to regain control during program execution failures.

CROMEMCO MEMORY

Cromemco memory boards include 4K byte and 16K byte RAM boards. The 4K RAM is a static RAM while the 16K RAM is dynamic with a "transparent" (to the user's program) refresh. Both the 4K and 16K RAM boards have a bank-select feature which permits memory expansion to eight banks of 64K bytes. In addition to the RAM memory, Cromemco offers two types of PROM boards. A 16K PROM board holds sixteen 1K byte 2708 EPROMs when fully populated. The 2708 is an erasable PROM. Exposure to ultra-violet light erases the contents of the PROM and allows reprogramming. PROM programming is possible by use of another Cromemco board, the

BYTESAVER™ board. This board contains a maximum of eight 1K byte 2708 EPROMs. One of the PROMs contains software that controls the on-board PROM programmer hardware. Using the BYTESAVER, any of the other PROMs on the board may be programmed in a few seconds by copying a 1K "memory image" of the program, or data in memory.

OTHER CROMEMCO BOARDS

A T-UART™ board provides two serial I/O ports, two 8-bit parallel I/O ports, and ten independent programmable internal timers. Baud rates of 110 to 9600 baud are software selectable.

A Digital Interface Board provides digital-to-analog, analog-to-digital and parallel I/O output and input. Up to seven channels of analog inputs (−2.56V to +2.54V) may be input with the board and converted to eight bits of digital data. The same number of channels are provided for analog outputs (−2.56V to +2.54V) which are converted from 8-bit digital data. In addition to the a-d and d-a conversions, eight bits of data may be input, or output, to a separate port for other control applications. The Digital Interface Board allows a Z-1, or Z-2, to interface to a variety of control applications, including real-time process control. Conversions are 5.5 μs, which are quite fast for a card of this kind.

Other Cromemco hardware are the TV DAZZLER™, which provides color graphics capability using an ordinary television, wire-wrap boards, extender cards, and a "joystick" console for analog-control inputs.

CROMEMCO Z-1 AND Z-2 MICROCOMPUTER SYSTEMS

The Z-1 Microcomputer System includes the Z-1 chassis with power supply (28 amp), twenty-two S-100 bus type card slots, and front control panel. The control panel allows an operator to examine and modify memory locations, perform a reset or external clear function, single step the CPU a cycle at a time, or execute the program from a given location. Indicator lights display the address and data-bus contents and system status.

The Z-1 includes the Z-80 CPU board, two 4K static RAM boards, an 8K PROM (BYTESAVER™) board with eight 2708 EPROMs, a resident monitor in PROM, and an RS-232 serial input/output interface.

The Z-2 Computer System in kit form includes the Z-2 chassis with 30-Amp power supply, one card socket and guide, front panel, and CPU board. In assembled form, the Z-2 includes the above items plus all 21 card sockets and guides and a cooling fan.

CROMEMCO SOFTWARE

The Cromemco 1K monitor includes commands to examine and modify memory, move and compare blocks of memory, read and load paper tapes, and program 2704 and 2708 PROMs using the BYTESAVER™ board. The monitor is supplied in the Z-1 system.

A resident 8K assembler is available in PROM, or on paper tape. A BASIC interpreter is available in PROM, or on paper tape. This BASIC is a special CONTROL BASIC that occupies 3K of memory and is designed for control and automated testing applications. The CONTROL BASIC allows the use of strings, multiple files, and subroutines, and also allows files to be programmed in PROMs on a BYTESAVER™ board.

THE DIGITAL GROUP, INC.

The Digital Group offers a number of CPU boards, among them the Motorola 6800, MOS Technology 6502, 8080A, and Z-80. All CPU boards are interchangeable and system compatibility is maintained at the CPU board level. A variety of system boards, peripherals, and microcomputer system combinations are offered by the company. System boards include a parallel I/O board, 8K static RAM, TV Readout and Audio-Cassette Interfaces, 4K EPROM memory board, and a Color Graphics Board. Peripheral equipment is generally housed in cabinets that match the other system cabinetry in styling and color. Keyboards, tv monitors, cassette drives, and a matrix printer are available. As with the other Z-80 microcomputer manufacturers, combinations of the various system components are offered as complete systems.

DIGITAL GROUP Z-80 CPU BOARD

As the Digital Group equipment is not S-100 bus compatible, much of the CPU card can be devoted to logic other than that dedicated to S-100 bus conversion. The Digital Group Z-80 CPU Board contains the Z-80 microprocessor, 2K bytes of 500-nanosecond static RAM, and 256 bytes of EPROM (1702A) containing a bootstrap loader. The bootstrap loader would ordinarily be used to load a small operating system from cassette which has the capability of cassette read and write, keyboard entry, and tv display. The system clock on the CPU board runs at 2.5 MHz. Logic is provided on the board to support DMA operations and all three Z-80 interrupt modes. Single step, and power on, reset are provided externally to the board.

DIGITAL GROUP MEMORY BOARDS

Two types of 8K memory boards are available, both using 2102 static RAM memory chips. One version is a 500-nanosecond memory, while the other board is a low-power 250-nanosecond memory. No wait states are required for the Z-80 CPU with the 500-nanosecond version. A 1702A EPROM board holds 4K bytes of 1702A EPROM memory (16 chips as the 1702 is a 256-byte EPROM chip).

DIGITAL GROUP I/O INTERFACES AND DEVICES

An Input/Output Board provides four 8-bit input ports and four 8-bit latching output ports. The I/O board supports either a memory-mapped I/O scheme as in the Motorola 6800 microprocessor, or the 8080/Z-80 I/O mapped scheme. (In the memory-mapped scheme, I/O devices are not addressed by I/O instructions but as 16-bit addresses).

A TV Readout and Audio-Cassette Interface Board allows both a character-oriented tv display and audio-cassette recording and playback. The TV Readout portion of the board provides a 64-character by 16-line display with a 7- by 9-dot matrix display of characters. The board will store the 1K characters in on-board RAM storage. One hundred twenty eight ASCII characters are displayable including both upper- and lower-case alphabets, numbers, extended math symbols, and Greek alphabet. The cursor may be positioned forward and backward under software control. Output of the tv section of the board is to a standard monitor (Digital Group or others) or to a standard television with input to the video section.

The Audio-Cassette portion of the board records, or reads, data on standard Phillips-type audio-cassette recorders using FSK (frequency-shift keying) recording. Data rates are 1100 baud which is the equivalent of 100 characters per second, approximately 10 times faster than a standard teletypewriter.

A separate Cassette-Storage System is also available, using one to four Phi-Deck cassette transports. The Phi-Deck is a quality audio-cassette drive with a file-search capability and other functions. Data rates are 800 bytes per second (!) and a search speed of 100 inches per second. A software operating system (driver) is supplied to implement recording of multiple blocks, single blocks, reading, CRC check, fast reverse and forward, and block search. The Cassette-Storage System requires a Phi-F interface board, and either a two- or four-drive cabinet with transports.

Another peripheral and interface offered is the Digital Group 96-Column Impact Printer. This is a relatively inexpensive 120 character per second, 96 character per line printer that prints on

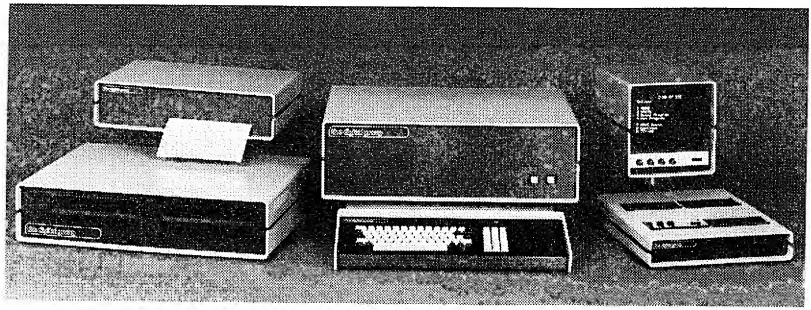
an 8½ inch page. The printer is a 5- by 7-dot matrix printer that prints 12 characters per inch horizontally and 6 lines per inch vertically. The associated interface card may be interfaced to existing I/O ports.

A 4096-pixel Color Graphics Board provides a color display of a 64 by 64 matrix. Three on-board 4K dynamic memories (red, green, and blue) result in eight different hues in any of the 4K pixels. The Color Graphics Board is driven by two 8-bit parallel I/O ports.

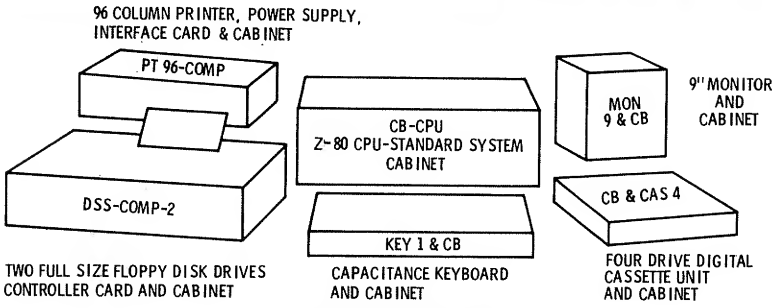
Other I/O equipment includes a 76-key ASCII keyboard with cabinet, a stand-alone cassette interface, prototyping boards and extenders, and miscellaneous hardware.

DIGITAL GROUP SYSTEMS

Various system combinations are offered by The Digital Group ranging from a four-board Z-80 system with 10K of RAM, I/O boards, and TV Readout/Audio-Cassette Interface, to a "hard copy" system with Z-80 CPU, TV Readout/Audio-Cassette Interface, two I/O Boards, 18K of RAM, Cassette Drive Interface with four Cas-



(A) Table-top arrangement.



(B) Cabinet identifier.

Courtesy The Digital Group, Inc.

Fig. 18-6. The Digital Group System 7—the ultimate.

sette drives, Monitor, Keyboard, and 96-Column Printer like the one shown in Fig. 18-6A. All of the systems are of the "box without control panel" type and include all cabinetry, cabling, power supplies, and so forth to make up a turn-key system. Virtually all system components and complete systems may be obtained in kit, or fully assembled, form.

DIGITAL GROUP SOFTWARE

Software available from The Digital Group includes a Cassette Storage Operating System, TINY BASIC, MAXI-BASIC, a Text-Editor, Z-80 Assembler, Z-80 Disassembler, and a variety of other applications programs and games.

The Cassette Storage Operating System, PHIMON, requires about 3K of memory. It is essentially a file-manage package operating with the Phi-Deck cassette transports. Programs may be loaded or saved on tape as files, and system utility files such as Debugging Software may be loaded in for execution. Complete file-manage functions, such as display and updates of tape directories are available to the user.

A small subset of BASIC, TINY BASIC, and MAXI-BASIC are available as higher-level interpretive packages. MAXI-BASIC for the Z-80 provides bcd floating-point arithmetic, formatted output, multiple statements per line, and multiple-line functions. Arrays may be any number of dimensions and string manipulation is provided. MAXI-BASIC requires 8K of memory and the recommended system configuration is 18K.

The Z-80 Assembler is a two-pass assembler requiring 12K plus working storage. The recommended system configuration is 18K.

RADIO SHACK

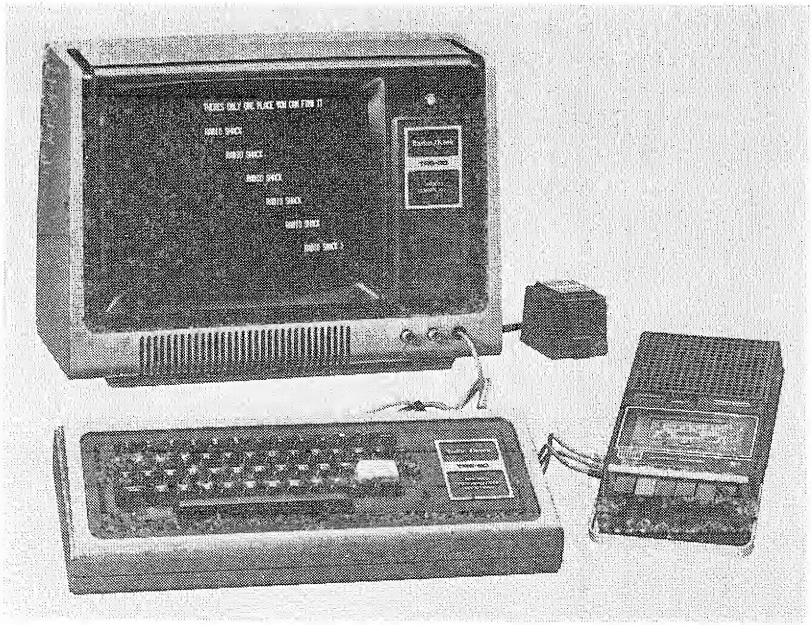
Radio Shack offers a Z-80 microcomputer system called the TRS-80. This system is unique from any other Z-80 system discussed in previous paragraphs as it is a completely integrated turnkey system (in the basic configuration). The user can purchase the system, take it home, plug it in, and can immediately compile his own BASIC program or run a predefined Radio Shack applications program!

RADIO SHACK HARDWARE

The basic TRS-80 system is shown in Fig. 18-7. It consists of a single-board computer enclosed within a cabinet that also contains the 53-key ASCII keyboard. A 12-inch monitor is used as a video display; 16 lines of 64 characters-per-line may be displayed with

automatic scrolling. In addition, graphics capability of 128 horizontal by 48 vertical elements is provided. A cassette recorder for secondary storage is included in the system. Transfer rates to the recorder are at 250 baud.

Memory in the TRS-80 consists of 4K bytes of ROM and 4K bytes of dynamic RAM. The ROM is dedicated to a BASIC interpreter, keyboard scanning routines, and drivers for the video display and cassette.



Courtesy Radio Shack, a Tandy Corp. Co.

Fig. 18-7. The Radio Shack TRS-80 microcomputer system.

The basic TRS-80 system may be expanded with up to 62K bytes of additional memory. Within the keyboard case up to 12K of ROM and 16K of RAM may be added. An "expansion module" is optionally provided for memory above these limits.

Although the system includes all cabling and connections for the video display and cassette recorder, additional I/O devices may be added by means of a 40-pin external connector on the rear of the cabinet. The Z-80 address bus, data bus, input/output, read, write interrupt, and interrupt acknowledge signals are available on the connector. Line printers, a floppy disc, a serial I/O unit, and modem are some of the peripherals currently available or available in the near future.

RADIO SHACK SOFTWARE

The 4K version of Radio Shack BASIC includes floating-point arithmetic, numeric, array, and (limited) string manipulation. It also includes video-graphics commands and cassette save and load commands. The SET(x,y) command turns on graphics point x,y, RESET (x,y) turns point x,u off, and POINT (x,y) determines the state of the point. CLS clears the screen. Direct cursor control is provided with the PRINT AT(x) command.

The 4K version of BASIC will be supplemented by a 12K version. Other software, such as editors, assemblers, and disc operating systems are planned. In addition, a wide range of applications software, such as game programs and business applications packages are currently available.

APPENDIX A

Z-80 Electrical Specifications

Z-80 electrical specifications, Z-80 CPU ac characteristics, and ac timing diagram are provided in the tables and figures on the following pages.

Electrical Specifications

ABSOLUTE MAXIMUM RATINGS

Temperature Under Bias	0°C to 70°C
Storage Temperature	−65°C to +150°C
Voltage on Any Pin with Respect to Ground	−0.3V to +7V
Power Dissipation	1.1W

*Comment

Stresses above those listed under "Absolute Maximum Rating" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other condition above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

DC CHARACTERISTICS

$T_A = 0°C$ to $70°C$, $V_{cc} = 5V \pm 5\%$ unless otherwise specified

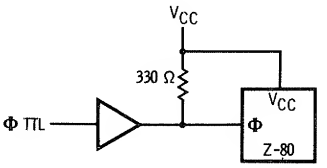
Symbol	Parameter	Min.	Typ.	Max.	Unit	Test Condition
V_{ILC}	Clock Input low Voltage	−0.3		0.45	V	
V_{IHC}	Clock Input High Voltage	V_{cc}[1]		V_{cc}	V	
V_{IL}	Input Low Voltage	−0.3		0.8	V	
V_{IH}	Input High Voltage	2.0		V_{cc}	V	
V_{OL}	Output Low Voltage			0.4	V	$I_{OL} = 1.8mA$
V_{OH}	Output High Voltage	2.4			V	$I_{OH} = -100\mu A$
I_{CC}	Power Supply Current			200	mA	$t_c = 400nsec$
I_{LI}	Input Leakage Current			10	μA	$V_{IN} = 0$ to V_{cc}
I_{LOH}	Tri-State Output Leakage Current in Float			10	μA	$V_{OUT} = 2.4$ to V_{cc}
I_{LOL}	Tri-State Output Leakage Current in Float			−10	μA	$V_{OUT} = 0.4V$
I_{LD}	Data Bus Leakage Current in Input Mode			±10	μA	$0 \leqslant V_{IN} \leqslant V_{cc}$

CAPACITANCE

$T_A = 25°C$, $f = 1$ MHz

Symbol	Parameter	Typ.	Max.	Unit	Test Conditions
C_Φ	Clock Capacitance		20	pF	Unmeasured Pins Returned to Ground
C_{IN}	Input Capacitance		5	pF	
C_{OUT}	Output Capacitance		10	pF	

Fig. A-1. Clock driver.



An external clock pull-up resistor of (330Ω) will meet both ac and dc clock requirements.

Z-80 CPU AC Characteristics

$T_A = 0°C$ to $70°C$, $V_{cc} = +5V \pm 5\%$, Unless Otherwise Noted.

Signal	Symbol	Parameter	Min	Max	Unit	Test Condition
Φ	t_c	Clock Period	.4	[12]	μsec	
	$t_w(\Phi H)$	Clock Pulse Width, Clock High	180	∞	nsec	
	$t_w(\Phi L)$	Clock Pulse Width, Clock Low	180	2000	nsec	
	$t_{r,f}$	Clock Rise and Fall Time		30	nsec	
A_{0-15}	$t_{(D)AD}$	Address Output Delay		160	nsec	$C_L = 100pF$
	$t_{F(AD)}$	Delay to Float		110	nsec	
	t_{acm}	Address Stable Prior to $\overline{MREQ}$ (Memory Cycle)	[1]		nsec	
	t_{aci}	Address Stable Prior to $\overline{IORQ}$, $\overline{RD}$ or $\overline{WR}$ (I/O Cycle)	[2]		nsec	
	t_{ca}	Address Stable From $\overline{RD}$ or $\overline{WR}$	[3]		nsec	
	t_{caf}	Address Stable From $\overline{RD}$ or $\overline{WR}$ During Float	[4]		nsec	
D_{0-7}	$t_{D(D)}$	Data Output Delay		260	nsec	$C_L = 200pF$
	$t_{F(D)}$	Delay to Float During Write Cycle		90	nsec	
	$t_{S\Phi(D)}$	Data Setup Time to Rising Edge of Clock During M1 Cycle	50		nsec	
	$t_{S\bar{\Phi}(D)}$	Data Setup Time to Falling Edge of Clock During M2 to M5	60		nsec	
	t_{dcm}	Data Stable Prior to $\overline{WR}$ (Memory Cycle)	[5]		nsec	
	t_{dci}	Data Stable Prior to $\overline{WR}$ (I/O Cycle)	[6]		nsec	
	t_{cdf}	Data Stable From $\overline{WR}$	[7]		nsec	
	t_H	Any Hold Time for Setup Time	0		nsec	
MREQ	$t_{DL\bar{\Phi}(MR)}$	$\overline{MREQ}$ Delay From Falling Edge of Clock, $\overline{MREQ}$ Low		100	nsec	$C_L = 50pF$
	$t_{DH\Phi(MR)}$	$\overline{MREQ}$ Delay From Rising Edge of Clock, $\overline{MREQ}$ High		100	nsec	
	$t_{DH\bar{\Phi}(MR)}$	$\overline{MREQ}$ Delay From Falling Edge of Clock, $\overline{MREQ}$ High		100	nsec	
	$t_{w(\overline{MRL})}$	Pulse Width, $\overline{MREQ}$ Low	[8]		nsec	
	$t_{w(\overline{MRH})}$	Pulse Width, $\overline{MREQ}$ High	[9]		nsec	

IORQ	$t_{DL\Phi(IR)}$	$\overline{IORQ}$ Delay From Rising Edge of Clock, $\overline{IORQ}$ Low		90	nsec	$C_L = 50pF$
	$t_{DL\Phi(IR)}$	$\overline{IORQ}$ Delay From Falling Edge of Clock, $\overline{IORQ}$ Low		110	nsec	
	$t_{DH\Phi(IR)}$	$\overline{IORQ}$ Delay From Rising Edge of Clock, $\overline{IORQ}$ High		100	nsec	
	$t_{DH\Phi(IR)}$	$\overline{IORQ}$ Delay From Falling Edge of Clock, $\overline{IORQ}$ High		110	nsec	
RD	$t_{DL\Phi(RD)}$	$\overline{RD}$ Delay From Rising Edge of Clock, $\overline{RD}$ Low		100	nsec	$C_L = 50pF$
	$t_{DL\Phi(RD)}$	$\overline{RD}$ Delay From Falling Edge of Clock, $\overline{RD}$ Low		130	nsec	
	$t_{DH\Phi(RD)}$	$\overline{RD}$ Delay From Rising Edge of Clock, $\overline{RD}$ High		100	nsec	
	$t_{DH\Phi(RD)}$	$\overline{RD}$ Delay From Falling Edge of Clock, $\overline{RD}$ High		110	nsec	
WR	$t_{DL\Phi(WR)}$	$\overline{WR}$ Delay From Rising Edge of Clock, $\overline{WR}$ Low		80	nsec	$C_L = 50pF$
	$t_{DL\Phi(WR)}$	$\overline{WR}$ Delay From Falling Edge of Clock, $\overline{WR}$ Low		90	nsec	
	$t_{DH\Phi(WR)}$	$\overline{WR}$ Delay From Falling Edge of Clock, $\overline{WR}$ High		100	nsec	
	$t_{W(WRL)}$	Pulse Width, $\overline{WR}$ Low	[10]		nsec	
M1	$t_{DL(M1)}$	$\overline{M1}$ Delay From Rising Edge of Clock, $\overline{M1}$ Low		130	nsec	$C_L = 30pF$
	$t_{DH(M1)}$	$\overline{M1}$ Delay From Rising Edge of Clock, $\overline{M1}$ High		130	nsec	
RFSH	$t_{DL(RF)}$	$\overline{RFSH}$ Delay From Rising Edge of Clock, $\overline{RFSH}$ Low		180	nsec	$C_L = 30pF$
	$t_{DH(RF)}$	$\overline{RFSH}$ Delay From Rising Edge of Clock, $\overline{RFSH}$ High		150	nsec	
WAIT	$t_{s(WT)}$	$\overline{WAIT}$ Setup Time to Falling Edge of Clock	70		nsec	
HALT	$t_{D(HT)}$	$\overline{HALT}$ Delay Time From Falling Edge of Clock		300	nsec	$C_L = 50pF$
INT	$t_{s(IT)}$	$\overline{INT}$ Setup Time to Rising Edge of Clock	80		nsec	
NMI	$t_{w(NML)}$	Pulse Width, $\overline{NMI}$ Low	80		nsec	
BUSRQ	$t_{s(BQ)}$	$\overline{BUSRQ}$ Setup Time to Rising Edge of Clock	80		nsec	
BUSAK	$t_{DL(BA)}$	$\overline{BUSAK}$ Delay From Rising Edge of Clock, $\overline{BUSAK}$ Low		120	nsec	$C_L = 50pF$
	$t_{DH(BA)}$	$\overline{BUSAK}$ Delay From Falling Edge of Clock, $\overline{BUSAK}$ High		110	nsec	
RESET	$t_{s(RS)}$	$\overline{RESET}$ Setup Time to Rising Edge of Clock	90		nsec	
	$t_{F(C)}$	Delay to Float ($\overline{MREQ}$, $\overline{IORQ}$, ($\overline{RD}$ and $\overline{WR}$)		100	nsec	
	t_{mr}	$\overline{M1}$ Stable Prior to $\overline{IORQ}$ (Interrupt Ack.)	[11]		nsec	

Z-80 CPU AC Characteristics—cont

Notes

[1] $t_{acm} = t_{w(\Phi H)} + t_f - 75$
[2] $t_{act} = t_c - 80$
[3] $t_{ca} = t_{w(\Phi L)} + t_r - 40$
[4] $t_{caf} = t_{w(\Phi L)} + t_r - 60$
[5] $t_{dcm} = t_c - 180$
[6] $t_{dct} = t_{w(\Phi L)} + t_r - 180$
[7] $t_{cdf} = t_{w(\Phi L)} + t_r - 50$
[8] $t_{w(\overline{MRL})} = t_c - 40$
[9] $t_{w(\overline{MRH})} = t_{w(\Phi H)} + t_f - 30$
[10] $t_{w(WR)} = t_c - 40$
[11] $t_{mr} = 2t_c + t_{w(\Phi H)} + t_f - 80$
[12] $t_c = t_{w(\Phi H)} + t_{w(\Phi L)} + t_r + t_f$

NOTES:

1. Data should be enabled onto the CPU data bus when $\overline{RD}$ is active. During interrupt acknowledge data should be enabled when $\overline{M1}$ and $\overline{IORQ}$ are both active.
2. All control signals are internally synchronized, so they may be totally asynchronous with respect to the clock.
3. The $\overline{RESET}$ signal must be active for a minimum of 3 clock cycles.
4. Output Delay vs. Loaded Capacitance
   TA = 70°C Vcc = +5V ±5%
   (1) $\Delta C_L = +100pF$ ($A_\phi - A_{15}$ and Control Signals), add 30 ns to timing shown.
   (2) $\Delta C_L = -50pF$ ($A_\phi - A_{15}$ and Control Signals), subtract 15 ns from timing shown.

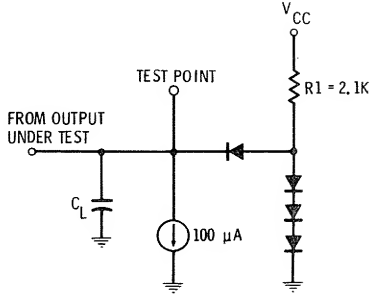


Fig. A-2. Load circuit for output.

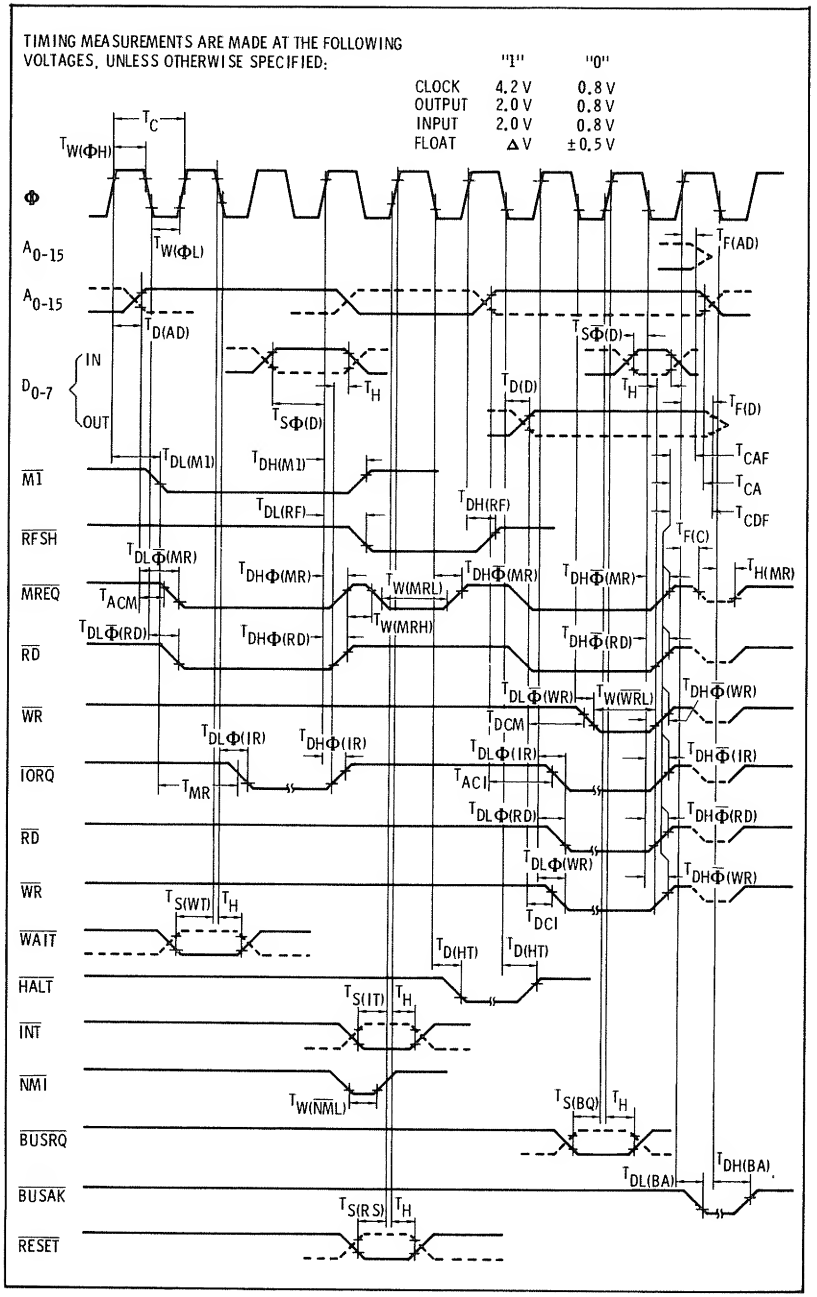


Fig. A-3. Z-80 ac timing diagram.

APPENDIX B

8080 and Z-80 Instructions Compared

Table B-1. 8080 and Z-80 Instructions

8080 Mnemonic	Z-80 Mnemonic	8080 Mnemonic	Z-80 Mnemonic	8080 Mnemonic	Z-80 Mnemonic
ACI	ADC A,N	IN	IN A,(N)	POP H	POP HL
ADC M	ADC A,(HL)	INR M	INC (HL)	POP PSW	POP AF
ADC r	ADC A,R	INR r	INC R	PUSH B	PUSH BC
ADD M	ADD A,(HL)	INX B	INC BC	PUSH D	PUSH DE
ADD r	ADD A,R	INX D	INC DE	PUSH H	PUSH HL
ADI	ADD A,N	INX H	INC HL	PUSH PSW	PUSH AF
ANA M	AND (HL)	INX SP	INC SP	RAL	RLA
ANA r	AND R	JC	JP C,NN	RAR	RRA
ANI	AND N	JM	JP M,NN	RC	RET C
CALL	CALL NN	JMP	JP NN	RET	RET
CC	CALL C,NN	JNC	JP NC,NN	RLC	RLCA
CM	CALL M,NN	JNZ	JP NZ,NN	RM	RET M
CMA	CPL	JP	JP P,NN	RNC	RET NC
CMC	CCF	JPE	JP PE,NN	RNZ	RET NZ
CMP M	CP (HL)	JPO	JP PO,NN	RP	RET P
CMP r	CP R	JZ	JP Z,NN	RPE	RET PE
CNC	CALL NC,NN	LDA	LD A,(NN)	RPO	RET PO
CNZ	CALL NZ,NN	LDAX B	LD A,(BC)	RRC	RRCA
CP	CALL P,NN	LDAX D	LD A,(DE)	RST	RST P
CP E	CALL PE,NN	LHLD	LD HL,(NN)	RZ	RET Z
CPI	CP N	LXI B	LD BC,NN	SBB M	SBC A,(HL)
CPO	CALL PO,NN	LXI D	LD DE,NN	SBB r	SBC A,R
CZ	CALL Z,NN	LXI H	LD HL,NN	SBI	SBC A,N
DAA	DAA	LXI SP	LD SP,NN	SHLD	LD (NN),HL
DAD B	ADD HL,BC	MVI M	LD (HL),N	SPHL	LD SP,HL
DAD D	ADD HL,DE	MVI r	LD R,N	STA	LD (NN),A
DAD H	ADD HL,HL	MOV M,r	LD (HL),R	STAX B	LD (BC),A
DAD SP	ADD HL,SP	MOV r,M	LD R,(HL)	STAX D	LD (DE),A
DCR M	DEC (HL)	MOV r1,r2	LD R,R'	STC	SCF
DCR r	DEC R	NOP	NOP	SUB M	SUB (HL)
DCX B	DEC BC	ORA M	OR (HL)	SUB r	SUB R
DCX D	DEC DE	ORA r	OR R	SUI	SUB N
DCX H	DEC HL	ORI	OR N	XCHG	EX DE,HL
DCX SP	DEC SP	OUT	OUT (N),A	XRA M	XOR (HL)
DI	DI	PCHL	JP (HL)	XRA r	XOR R
EI	EI	POP B	POP BC	XRI	XOR N
HLT	HALT	POP D	POP DE	XTHL	EX (SP),HL

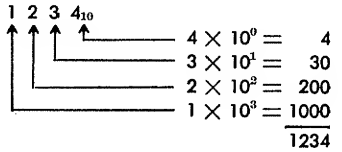
APPENDIX D

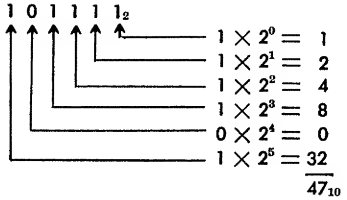
Binary and Hexadecimal Representation

BINARY AND HEXADECIMAL REPRESENTATION

Binary Representation

In binary, positional notation is used similarly to decimal notation:

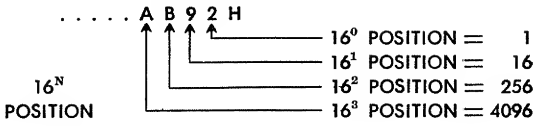




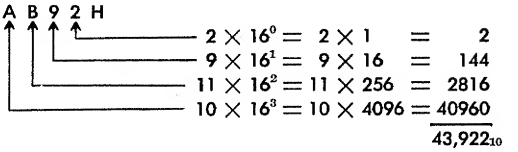
Hexadecimal Representation

$Decimal_{10}$	$Binary_2$	$Hexadecimal_{16}$
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Hexadecimal to Decimal Conversion



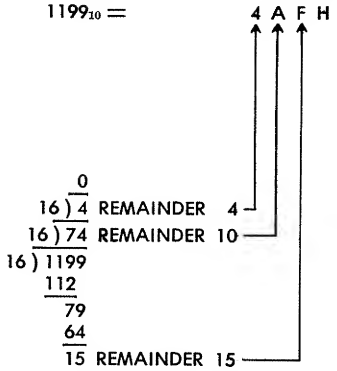
To convert:



Steps in Conversion:

1. Multiply each digit weight by hex digit.
2. Add to get total equal to equivalent decimal #.

Decimal to Hexadecimal Conversion



Steps in Conversion:

1. Divide decimal # by 16.
2. Save remainder.
3. Repeat until quotient = 0.
4. Remainders in reverse order represent hexadecimal equivalent.

Hexadecimal Addition and Subtraction

		OPERAND 2														
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
	0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
	2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
	3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
	4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
	5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
OPERAND	**6**	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
1	**7**	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
	8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
	9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
	A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
	B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
	C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
	D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
	E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
	F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

To add operand 1 to operand 2 find sum at intersection. If two digits, a carry to high order is represented.

To subtract operand 1 from operand 2 find difference in operand 2 column. If operand 2 greater than one digit the high-order 1 represents a borrow from next digit.

APPENDIX E

ASCII Character Code

The ASCII character code is shown in Chart E-1 on the following page.

Chart E-1. ASCII Character Code

MOST SIGNIFICANT HEX DIGIT	LEAST SIGNIFICANT HEX DIGIT															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	−	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

APPENDIX F

Z-80 Microcomputer Manufacturers

Computer Systems
26401 Harper Avenue
St. Clair Shores, Michigan 48081

Cromemco, Inc.
2432 Charleston Road
Mountain View, California 94043

The Digital Group, Inc.
P. O. Box 6528
Denver, Colorado 80206

MiniMicroMart, Inc.
1618 James Street
Syracuse, New York 13203

Quay Corporation
P.O. Box 386
Freehold, New Jersey 07728

Radio Shack, A Division of
Tandy Corporation
One Tandy Center
Fort Worth, Texas 76102

S. D. Sales Company
P.O. Box 28810K
Dallas, Texas 75228

Technical Design Labs, Inc.
Research Park
Bldg. H
1101 State Road
Princeton, New Jersey 08540

Zilog, Inc.
10460 Bubb Road
Cupertino, California 95014

Index

the Z-80 microcomputer handbook

The Z-80 microcomputer represents, for the time being, the "state-of-the-art" in microcomputer technology. And until a successor makes its appearance, the Z-80 will doubtlessly be attractive to many computer users. It is the author's intention to provide both the current user and the prospective user of microcomputers with a handbook that is beneficial to the technology. With this view in mind, the book will acquaint the reader with the hardware of the Z-80, present the multitudinous (in number of instructions) software aspects of the Z-80, and describe the other microcomputer systems built around the Z-80.

Organizationally, the book is arranged in three sections. Section I discusses Z-80 hardware. Chapters 2 and 3 look at architecture, interface signals, and timing. Chapters 4 and 5 handle addressing modes and instructions. In Chapter 6 is presented the effect of arithmetic operations and other operations on CPU flags. The next chapter contains the powerful interrupt sequences of the Z-80. Chapter 8 describes interfacing examples of I/O on memory devices.

Section II presents the Z-80 software with a representative assembler program being introduced in the first chapter of the section. Machine language aspects are covered, as well. Chapters 10 through 15 detail the common programming operations of moving data, arithmetic operations, list and table procedures, subroutine use, and I/O functions relative to instruction set groups. Many examples of each operation are provided. The last chapter in the section offers some commonly used subroutines written in Z-80 assembly language.

Section III discusses microcomputers built around the Z-80. Chapter 17 presents the Zilog products including the microcomputer board products in the Z-80 family and development systems. In the last chapter, four other Z-80 microcomputer manufacturers' hardware and software are described: Technical Design Labs, Inc., Cromemco, Inc., the Digital Group, Inc., and Radio Shack.

The reader will benefit from this informative book and will receive valuable assistance in solving some of the hardware and software implementation problems.

ABOUT THE AUTHOR

William Barden, Jr., is currently a member of the Technical Staff of Development Laboratories, Inc., in Santa Monica, California. He has 16 years experience in technical writing, computer programming, computer design, and computer systems design, mostly making microcomputers. He has worked on more than a dozen different computer systems and has written articles on computer and related equipment for various publications. He is a member of the Association for Computing Machinery and the IEEE. His major interest is home computer systems. Mathematical games and sailing are among his other interests. Other SAMS books by William Barden are *How to Buy & Use Minicomputers & Microcomputers* and *How to Program Microcomputers*.

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

$8.95 / 21500

ISBN: 0-672-21500-4